aspecten van programmeertalen

ir. J.J. van AMSTEL
ir. J.A.A.M. POIRTERS

ACADEMIC SERVICE

ASPECTEN
VAN
PROGRAMMEERTALEN

aspecten van programmeertalen

ir. J.J. van AMSTEL
ir. J.A.A.M. POIRTERS

Academic Service
den haag 1983

Uitgegeven door: Academic Service
Postbus 96996
2509 JJ Den Haag
Zetwerk: mevr. I. Geerling-Engelbarts
Ontwerp omslag: JAM Gauw
Druk: Krips Repro Meppel
Bindwerk: Meeuwis, Amsterdam
ISBN: 90 6233 092 4

VOORWOORD

Er zijn veel programmeurs die een bepaalde programmeertaal volledig beheersen, maar die nauwelijks kennis hebben van constructiemogelijkheden en hun semantiek in andere programmeertalen. Ook het theoretische fundament ontbreekt veelal. Deze tekst is bedoeld om een aantal fundamentele concepten en algemene principes van veelgebruikte programmeertalen te belichten. Niet voor iedereen zullen alle onderwerpen en de diepgang van behandeling van belang zijn. Het boek is dan ook niet op de eerste plaats bedoeld voor zelfstudie. Een leraar of begeleider moet kunnen aanwijzen welke onderwerpen en welke diepgang in een gegeven situatie van belang zijn. Er wordt vanuit gegaan dat de lezer een redelijke ervaring in het programmeren heeft. Een combinatie van deze stof met het onderwijs in het programmeren lijkt echter ook zeer geschikt.

De tekst bestaat uit drie delen (plus een inleiding en een toevoegsel: hoofdstuk 10).

In het eerste deel (de hoofdstukken 1, 2, 3, 4 en 5) worden de begrippen syntaxis, semantiek en pragmatiek aan de orde gesteld. Bij de syntaxis worden onderwerpen als formele talen en automaten behandeld. Bij de semantiek worden korte inleidingen gegeven op drie manieren van semantiekbeschrijving.

Het tweede deel (de hoofdstukken 6 en 7) beschrijft algemene eigenschappen van programmeertalen en geeft mogelijke constructies (en hun effect) in programmeertalen aan.

Het derde deel (de hoofdstukken 8 en 9) geeft een beschrijving van enkele programmeertalen. In hoofdstuk 8 zijn dat enige algemene programmeertalen (Pascal, FORTRAN en COBOL). Van deze talen wordt, onder de veronderstelling dat van het voorgaande kennis is genomen, een redelijk uitvoerige beschrijving gegeven, zonder dat de tekst als vervanger van een manual in aanmerking kan komen. In hoofdstuk 9 komen enige speciale programmeertalen aan de orde (LISP, SNOBOL, APL en SIMULA), door enige karakteristieke constructiemogelijkheden te bespreken.

In de Inleiding worden de belangrijkste begrippen met betrekking tot programmeren en programmeertalen geïntroduceerd.

Hoofdstuk 10 beschrijft enige min of meer recente ontwikkelingen op het gebied van programmeren en programmeertalen.

Wij danken mevr. I. Geerling die het typewerk voor dit boek op voortreffelijke wijze verzorgde.

Voor eventuele op- of aanmerkingen houden wij ons ten zeerste aanbevolen.

Eindhoven, oktober 1982

J.J. van Amstel
J.A.A.M. Poirters

INHOUDSOPGAVE

INLEIDING

Onder *programmeren* worden wel de activiteiten verstaan die nodig zijn om een probleem uit een bepaald vakgebied op te lossen met behulp van een rekenmachine. Eén van de bedoelde activiteiten – vaak bedoelt men deze als men het over programmeren heeft – is het opstellen van de rij opdrachten die door de rekenmachine moet worden uitgevoerd. De notatie, waarin het handelingenvoorschrift (de rij opdrachten) moet worden vastgelegd, wordt *programmeertaal* genoemd. Op de ontwikkeling van programmeertalen hebben drie zaken een grote invloed gehad: de rekenmachine, het probleemgebied en het programmeren.

De invloeden van de rekenmachine en van het probleemgebied zijn vanaf het begin erg groot geweest bij de ontwikkeling van programmeertalen. Pas de laatste jaren wordt bij de definitie van nieuwe programmeertalen echt met de programmeur rekening gehouden, waarbij twee aspecten centraal staan: de correctheid en de duidelijkheid van programma's. In de 'klassieke' hogere programmeertalen zoals FORTRAN, ALGOL 60 en PL/1 zijn bijvoorbeeld al wel mogelijkheden voor modularisatie aanwezig, maar niet met het oog op de correctheid van programma's, wat nu als belangrijkste doel wordt gezien. Bij nieuwe programmeertalen zoals Pascal en Euclid is juist terdege rekening gehouden met de mogelijkheid om de correctheid van programma's aan te tonen. En talen als CLU en Alphard zijn gedefinieerd met als centraal thema de modularisatie.

Maar nu eerst de relatie tussen programmeren en de (hogere) programmeertalen.

Voor het beschrijven van een complex geheel staan ons twee vormen van beschrijving ter beschikking die we de *procesbeschrijving* en de *toestandsbeschrijving* kunnen noemen. Stel bijvoorbeeld dat een cirkel beschreven moet worden. Dit kan met de procesbeschrijving: "Om een cirkel te construeren draait men een, met één been vaststaande, passer zolang rond tot het andere been het uitgangspunt weer bereikt.". Of met de toestandsbeschrijving: "Een cirkel is de verzameling van alle punten die een gelijke (gegeven) afstand hebben tot een gegeven punt.".

De eerste beschrijving geeft aan *hoe* een cirkel geconstrueerd moet worden, de tweede beschrijving geeft aan *wat* een cirkel is. De meeste programmeertalen zijn talen waarin procesbeschrijvingen gegeven kunnen worden; er zijn echter ook ontwikkelingen om tot programmeertalen te komen waarin via toestandsbeschrijvingen een probleem opgelost kan worden. Bij het programmeren wordt de toestandsbeschrijving als hulpmiddel gebruikt om tot een juiste procesbeschrijving te komen.

Uitvoering van een programma leidt tot operaties op data. Een basisbegrip hierbij is het begrip *actie* (handeling).
Onder een actie verstaan we een gebeurtenis met een eindige tijdsduur, die een wel-gedefinieerd effect realiseert.
Om het effect van een actie te kunnen omschrijven gaan we er van uit dat deze actie zich afspeelt in een bepaalde *omgeving* en dat deze omgeving op ieder moment in een bepaalde *toestand* verkeert. Een actie heeft dan een *toestandstransformatie* tot gevolg van de toestand vóór de actie naar de toestand ná de actie. De omgeving, waarin een actie plaatsvindt, wordt gekenmerkt door een aantal, voor de actie relevante *grootheden*. Een toestand op een bepaald moment wordt dan bepaald door de op dat moment geldende waarden van deze grootheden. De grootheden worden *variabelen* genoemd. Met een variabele is verbonden een voor deze variabele karakteristieke *waardenverzameling* met daarop gedefinieerde *operaties*. Zo'n waardenverzameling met zijn operaties wordt een *type* genoemd. We kunnen de begrippen toestand, toestandsbeschrijving en actie nu nader preciseren.
Een toestand wordt gekarakteriseerd door een rij waarden van variabelen. Een toestandstransformatie is de verandering van de waarde(n) van één of meer variabelen. Een actie is de oorzaak van de verandering van de waarde(n) van deze variabele(n).
Vaak kan men een actie opvatten als een rij *deelacties*; uitgaande van een bepaalde begintoestand leiden de achtereenvolgende toestandstransformaties van de deelacties (via de bijbehorende tussentoestanden) tot een eindtoestand die gelijk is aan de toestand die bereikt wordt als we de actie als één geheel beschouwen.

Als de actie beschouwd wordt als een rij deelacties, wordt de actie een *proces* genoemd. We hebben ons beperkt tot processen waarvan de acties na elkaar plaatsvinden (sequentiële processen). In het algemeen is elke actie op te splitsen in deelacties en is iedere actie op te vatten als deelactie van een omvattend proces. De acties die niet meer op te splitsen zijn in deelacties worden *elementaire acties* genoemd. Welke acties elementair zijn, wordt bepaald door de programmeertaal.

Het verloop van een proces is geheel vastgelegd door de achtereenvolgende toestanden (de opvolging van de waarden van de variabelen). De acties en processen zijn gebeurtenissen die zich in de tijd afspelen. De beschrijving van een actie wordt een *statement* genoemd; zo spreekt

men ook van een *algoritme* als de beschrijving van een proces. Een *programma* is een algoritme geschreven in een programmeertaal.

Een actie heeft tot gevolg dat de waarde van één of meer variabelen verandert. De meest elementaire actie is het toekennen van een waarde aan een variabele. De desbetreffende beschrijving wordt *assignment statement* genoemd. In deze statement komen de naam van een variabele en een denotatie van een waarde voor; deze laatste kan zijn: een constante, de naam van een variabele die een waarde heeft en in het algemeen een expressie. Voorbeelden van assignment statements:

```
k := 3                {na afloop heeft k de waarde 3}
k := k + 1            {de waarde van k is met 1 verhoogd}
k := m * n + p ↑ 7
```

Naast statements bestaan er in programmeertalen notaties om assignment statements op zinvolle wijze te combineren, zoals bijvoorbeeld om aan te geven dat een bepaalde statement slechts onder bepaalde voorwaarden mag worden uitgevoerd of dat een statement herhaald moet worden uitgevoerd. Bovendien moet kunnen worden aangegeven of statements respectievelijk na elkaar of tegelijkertijd mogen worden uitgevoerd. De (notatie-) hulpmiddelen die ons hiertoe in de programmeertalen ter beschikking staan, worden *besturingsstructuren* (of sequentiëringsstructuren) genoemd, of óók wel statements. Met deze besturingsstructuren wordt dus de volgorde beschreven waarin de afzonderlijke acties van een proces moeten plaatsvinden.

Bij een beperking tot sequentiële processen kunnen drie methoden tot sequentiëring onderscheiden worden. We zullen deze hier introduceren in een door ons gekozen notatie (zoals we dat ook voor de assignment statement hebben gedaan) en het effect aangeven in termen van de uit te voeren acties.

1. Concatenatie

Aaneenrijging van een aantal statements tot een geordende rij van statements.

Notatie: S1; S2

Voorbeeld: m := 4; n := m + 5

2. Selectie/Conditie

Deze structuur bestaat in twee varianten:

- Keuze uit twee alternatieven

  Notatie: <u>if</u> B <u>then</u> S1 <u>else</u> S2 <u>fi</u>

  Als aan de voorwaarde B wordt voldaan, wordt statement S1 uitgevoerd, zo niet dan S2.

  Voorbeeld: <u>if</u> a > b <u>then</u> max := a <u>else</u> max := b <u>fi</u>
- Voorwaardelijke uitvoering

  Notatie: <u>if</u> B <u>then</u> S <u>fi</u>

  Als aan de voorwaarde B wordt voldaan, dan wordt statement S uitgevoerd; zo niet, dan vindt er geen actie plaats.

  Voorbeeld: <u>if</u> x < 0 <u>then</u> x := -x <u>fi</u>

3. *Repetitie*
Het onder een bepaalde voorwaarde herhaald uitvoeren van een statement.
Notatie: while B do S od
Als aan de voorwaarde B wordt voldaan, wordt statement S uitgevoerd; als (dan nog) aan B wordt voldaan, wordt S (nogmaals) uitgevoerd; .....; als aan de voorwaarde niet wordt voldaan, vindt geen actie (meer) plaats.
Voorbeeld: k := 0;
while 2 ↑ k < n do k := k + 1 od

De S, S1 en S2 uit bovenstaande beschrijvingen kunnen elk weer een van de bovenstaande constructies zijn.

Iedere waarde in een programma is van een bepaald type. Een type definieert een waardenverzameling tezamen met de operaties die op deze waarden mogelijk zijn. In een taal zijn meestal een aantal typen reeds gedefinieerd, bijvoorbeeld *integer* (waardenverzameling: deelverzameling van de verzameling der gehele getallen; operaties: rekenkundige operaties), *real* (waardenverzameling: deelverzameling van de verzameling der reële getallen; operaties: rekenkundige operaties), *boolean* (waardenverzameling: de logische waarden {true, false}; operaties: conjunctie, disjunctie, ontkenning, enz.) en *char* (waardenverzameling: de beschikbare karakterset).

Met de vier genoemde basistypen kan elk proces beschreven worden. Het is echter wenselijk voor de programmeur de mogelijkheid te hebben eigen, nieuwe typen te definiëren. Enkele redenen hiervoor zijn:
- door eigen typen kan de waardenverzameling beperkt worden tot die waarden die werkelijk een rol spelen in het programma;
- de waarden kunnen beter afgestemd worden op de waarden zoals die voorkomen in het vakgebied, waarvoor het programma wordt geschreven;
- de operaties kunnen afgestemd worden op het bedoelde gebruik.

Sommige programmeertalen beschikken dan ook over mogelijkheden om nieuwe typen te introduceren.

Bij de definitie van zo'n nieuw type krijgt het type een naam en wordt de waardenverzameling aangegeven:
type nieuw = <waardenverzameling>
Hieronder volgen een aantal methoden om nieuwe typen te introduceren. Allereerst de *opsomming* (*enumeratie*) van de waardenverzameling:
type kleur = (rood, geel, groen, blauw)
Op de tweede plaats de beperking tot een deelverzameling (*interval*, *subrange*) van de waardenverzameling van een bekend type:
type teller = 2 .. 100

Daarnaast bestaan er een aantal mogelijkheden om zogenaamde gestructureerde typen te definiëren. Deze typen worden gedefinieerd

in termen van reeds bekende typen en een waarde van zo'n nieuw type is dan ook opgebouwd uit waarden (componenten genoemd) van deze reeds bekende typen. Bij een bepaalde *structureringsmogelijkheid* behoren vele typen; al deze typen hebben gemeen dat een waarde op dezelfde manier uit componenten is opgebouwd.

Als structureringsmogelijkheid kunnen bijvoorbeeld genoemd worden:

- *record* (het cartesisch produkt van waardenverzamelingen);
- *array* (afbeelding van een verzameling in een andere verzameling);
- *sequence* (verzameling van alle rijen over een verzameling);
- *set* (de waarde van een variabele is een verzameling).

Iedere variabele moet *gedeclareerd* worden; dit houdt in dat de naam van de variabele en het type van de variabele worden opgegeven:

var p : integer

Voordat een variabele gedeclareerd kan worden moet het desbetreffende type bekend zijn door definitie of doordat het type standaard in de taal is opgenomen. Na declaratie bestaat de variabele, maar de waarde is nog niet gedefinieerd.

Een toestand van een proces wordt bepaald door de waarden van de in het proces optredende variabelen. De typen van deze variabelen leggen tezamen vast welke toestanden in het proces mogelijk zijn. De verzameling van alle mogelijke toestanden wordt de *toestandsruimte* van het proces genoemd. De toestandsruimte van een proces is bepaald door de *declaratie van de variabelen*. De declaraties van de variabelen worden in het begin van de desbetreffende procesbeschrijving opgenomen, vóór de statements. Na afloop van een proces bestaat de toestandsruimte van het proces in het algemeen niet meer.

De declaraties worden onderling gescheiden door puntkomma's en ook wordt het declaratiegedeelte van de 'eigenlijke' procesbeschrijving gescheiden door een puntkomma.

Een *expressie* is een notatie voor een samengestelde operatie: een rij operanden (constanten, variabelen, expressies) onderling gescheiden door operatoren (eventueel worden nog haakjes gebruikt). De samenstellende operaties zijn de operaties van een type. Iedere operator verwacht operanden van het bijbehorende type en levert een resultaat af van een type dat behoort bij de operator (de optelling geeft een resultaat dat hetzelfde type heeft als de operanden, 3<7 levert een boolean resultaat).

Een expressie heeft een waarde die berekend kan worden als de variabelen een waarde hebben (gedefinieerd zijn). De waarde van de expressie is eenduidig bepaald als er een volgorde afgesproken is waarin de operaties worden uitgevoerd.

We hebben de elementaire statements (eigenlijk maar één: de assignment statement), de besturingsstructuren en de typen kort bekeken. We komen nu terug op het begrip proces.

Het *effect* van een actie is de transformatie van een begintoestand naar een eindtoestand. De begintoestand wordt gekarakteriseerd door de *invoerwaarden* (bij de assignment: de waarden die in de expressie in het rechterlid voorkomen) voor de actie.

De actie heeft als resultaat dat *uitvoervariabelen* (bij de assignment: de variabele in het linkerlid) een waarde hebben gekregen; deze uitvoerwaarden leggen de eindtoestand vast.

We bekijken nu niet-elementaire acties (deelprocessen) en hun (proces)beschrijving.

De beschrijving van een deelproces noemen we een *procedure* (of subroutine). De activering van de procedure vindt plaats door middel van de uitvoering van de bijbehorende *procedure statement*. Bij het gebruik van de statement zijn wij slechts geïnteresseerd in het effect van de actie (niet in het erbij behorende proces).

In een (procedure) statement moeten altijd de volgende drie componenten aanwezig zijn:
- de *identificatie* van de procedure; welke procedure wordt actief, welke actie vindt plaats;
- de *invoerwaarden*; op welke operanden wordt door de procedure geopereerd, hoe wordt de begintoestand vastgelegd;
- de *uitvoervariabelen*; waar zijn na afloop van de actie de resultaten terug te vinden.

VOORBEELD procedure statement
divmod(a,b,q,r)
De naam van de procedure is divmod. De invoerwaarden zijn a en b en de uitvoervariabelen zijn q en r. Dit is niet te zien aan de statement en moet elders zijn vastgelegd.
(einde voorbeeld)

Van een aantal acties (de elementaire acties uit de programmeertaal) behoeft de programmeur de procesbeschrijving niet te geven, ze zijn als het ware in de taal opgenomen. Van alle andere procedures moet de programmeur de beschrijving zelf maken en binnen zijn programma opnemen. Deze beschrijving binnen het programma wordt de *declaratie* van de procedure genoemd. Bij het ontwerpen (en declareren) van een procedure wordt gebruik gemaakt van reeds bekende procedures.

Bij de procedure spelen drie aspecten een rol:
1. Het *effect* van de procedure: wat bewerkstelligt de procedure.
2. De *invoerwaarden* en *uitvoervariabelen*: waarop werkt de procedure. In de procedure (declaratie) worden deze grootheden gerepresenteerd door formele namen, deze worden de *formele parameters* genoemd. Bij de activering van de procedure moeten actuele waarden en actuele uitvoervariabelen, samen de zogenaamde *actuele parameters*, bekend gemaakt worden. Deze actuele para-

meters zijn de vervangers voor de formele namen die in de proceduredeclaratie gebruikt worden.
De parameters leggen de relatie van de procedure met de omgeving vast.
Een procedure beschrijft een hele klasse van processen; wordt de procedure geactiveerd met andere invoerwaarden, dan wordt er een ander proces uitgevoerd.

3. De *beschrijving* van het proces, dit is het patroon van acties dat, uitgaande van de begintoestand (bepaald door de invoerwaarden), de eindtoestand (vastgelegd in de uitvoervariabelen) produceert. Dit algoritme kent zijn eigen toestandsruimte en in de acties spelen de variabelen uit deze toestandsruimte (*lokale variabelen*) en de parameters een rol. Deze eigenlijke procesbeschrijving noemt men wel de *procedure-body*.

In de proceduredeclaratie moeten worden vastgelegd:
- de naam van de procedure;
- de namen van de formele parameters en de typen hiervan;
- de body van de procedure.

Het effect van de procedure moet vastgelegd worden in een stuk documentatie.

VOORBEELD procedure declaratie

```
procedure divmod(deeltal,deler: in integer; quot,rest: out integer);
    begin quot := 0; rest := deeltal;
          while rest ≧ deler
            do rest := rest-deler;
               quot := quot + 1
            od
    end
```

(einde voorbeeld)

Activering van de procedure bestaat uit twee fasen:
1. Parameterinitialisering. Dit houdt in dat er voor de uitvoering van de body een relatie moet worden gelegd tussen de actuele en formele parameters.
2. Uitvoering van de body waarbij met de onder 1. genoemde relaties rekening wordt gehouden.

Als de bovenstaande procedure geactiveerd wordt door de statement divmod(a,b,q,r), wordt er eerst een relatie gelegd tussen a en deeltal, b en deler, q en quot en r en rest, daarna wordt de body uitgevoerd.

We kunnen nu aangeven hoe een programma in het algemeen is opgebouwd. Een *programma* bestaat uit:
- definitie van typen
- declaratie van de variabelen
- declaratie van de procedures
- het algoritme, d.i. de rij statements die het uit te voeren proces beschrijven (d.w.z. de toestandstransformatie realiseren).

In het bovenstaande zijn een aantal elementen aan de orde gekomen die in een programmeertaal aanwezig moeten zijn om de opdrachten tot acties en de waarden te kunnen noteren. Het niveau waarop we willen werken, dat wil zeggen de machtigheid van de acties, de ter beschikking staande besturingsstructuren en de mogelijkheden voor typering van waarden, bepalen of we een bepaalde taal kunnen gebruiken (even afgezien van het toepassingsgebied).

De meeste huidige talen zijn via een aantal stappen uit de mogelijkheden van de hardware ontstaan. Een aantal constructiemogelijkheden in hogere programmeertalen is dan ook nog steeds direct terug te voeren tot deze mogelijkheden in de hardware. We kunnen hierbij bijvoorbeeld denken aan sprongopdrachten (het veranderen van de waarde van de opdrachtteller) en het gebruik van pointers (het werken met adressen).

In de hardware is een grote ontwikkeling te zien geweest. Deze ontwikkeling is vooral gericht geweest op het verhogen van de snelheid, het vergroten van de performance en het verminderen van de kosten. Bijna nooit (een uitzondering is de Burroughs B5500 en zijn opvolgers) werd rekening gehouden met de programmeerbaarheid van de nieuwe hardware. Zelfs als er op een bepaald moment een goede programmeertaal bestond, waren er steeds weer ontwikkelingen die de programmeurs noodzaakten machinegericht te werken. Wat die ontwikkelingen betreft kunnen we denken aan multiprocessoren, minicomputers, microcomputers, netwerken, (gedistribueerde) databases en real-time toepassingen. Taalconstructies, aangepast aan deze ontwikkelingen, werden pas veel later in hogere programmeertalen opgenomen, omdat er bij de ontwikkeling van de hardware geen rekening gehouden is met de programmeerbaarheid ervan, anders dan op zeer laag niveau.

Bovendien worden de faciliteiten voor bijvoorbeeld parallelle processen en de input en de output vaak ingegeven door de mogelijkheden (of het gemak) tot implementatie in de hardware of het operating system, waarbij de programmeerbaarheid dan weer op de tweede plaats staat.

Het is duidelijk dat de hardware, het systeem, grote invloed heeft op wat in de taal aan constructiemogelijkheden aanwezig is.

De toepassing zal bepalen welke structuren in de taal aanwezig moeten zijn. De toepassingen zijn de laatste jaren nogal veranderd, zowel in aard als in omvang. We kunnen hierbij bijvoorbeeld denken aan de verschuiving van rekenintensieve toepassingen naar gegevensintensieve toepassingen. Dit heeft uiteraard consequenties voor de gewenste programmeertaal. We spreken wel van actie-georiënteerde en gegevens-georiënteerde programmeertalen. Een voorbeeld uit de eerste categorie is ALGOL 60, en COBOL is een voorbeeld uit de tweede categorie.

Vooral bij de eerste programmeertalen werd meer aandacht geschonken aan actie-georiënteerde constructies; de nadruk ligt op

het opstellen van het algoritme, waarbij het aantal benodigde gegevens klein is. Bij de andere categorie staan juist de gegevens centraal en is het algoritme daaraan ondergeschikt, ook al omdat dit meestal eenvoudig is. We kunnen hierbij denken aan administratieve toepassingen, maar ook aan andere toepassingen waar computers in een groter systeem zijn opgenomen.

Voor veel toepassingsgebieden zijn aparte talen ontworpen.

Naast de verschuiving van actie-georiënteerde naar gegevens-georiënteerde toepassingen is ook de verschuiving van batch-verwerking naar interactieve verwerking belangrijk geworden. De programma's zijn ook veel groter geworden en niet bedoeld om een enkele keer verwerkt te worden. Dit betekent dat de taal niet meer alleen gezien wordt als de notatie voor het algoritme, maar ook als middel om de complexiteit van een softwareproject te beheersen. Dit houdt in dat de taal ook beoordeeld wordt op aspecten als betrouwbaarheid, onderhoudbaarheid en bewijsbaarheid van de programma's die erin geschreven kunnen worden.

Programmeertalen zorgen voor abstractie, zowel van computers waarop ze geïmplementeerd worden als van de applicaties waarvoor ze bedoeld zijn. De programmeertaal brengt de toepassing en het computersysteem bij elkaar. Het verschil tussen die twee is vaak erg groot doordat de machine in principe op een laag niveau staat wat de programmeerbaarheid betreft.

Programmeertalen hebben net als natuurlijke talen een grote invloed op onze manier van denken. Huidige hogere programmeertalen verschaffen zeer goede hulp bij de probleemoplossing, maar tegelijkertijd vormen ze ook nog steeds een belemmering voor de wijze waarop we ons kunnen uitdrukken en dus voor de manier waarop we over de oplossing van een probleem kunnen denken. Langzamerhand echter krijgt dit laatste aspect meer aandacht van de taalontwerpers.

1 ONTWIKKELING VAN PROGRAMMEERTALEN

1.1 INLEIDING

In dit hoofdstuk gaan we in op de geschiedenis van programmeertalen. We beginnen bij de binaire code en eindigen bij de nieuwe programmeertalen Pascal en Ada. Van een aantal hogere talen worden enige karakteristieken beschreven.

1.2 MACHINETAAL; BINAIRE CODE

De machine kent zijn eigen, door de fabrikant ingebouwde, repertoire van elementaire acties. Iedere opdracht voor zo'n actie wordt in de machine gerepresenteerd door een bepaald bitpatroon, meestal ter lengte van een geheugenwoord. Voor veel machines is het zo, dat het woord dat de opdracht voorstelt uit twee delen bestaat: *functiedeel* en *adresdeel*. Het eerste deel, het functiedeel, specificeert de eigenlijke opdracht of operatie (bijvoorbeeld optellen, aftrekken of een transport van of naar het geheugen); het tweede deel, het adresdeel, specificeert het adres van het geheugenwoord waarvan de inhoud bij de opdracht is betrokken; dit tweede deel kan ook als constante optreden. We noemen het tweede deel ook wel het operanddeel, omdat de inhoud van het genoemde adres als operand in de opdracht optreedt. We beperken ons hier tot 1-adres opdrachten; er zijn ook machines die een 2-adres opdrachtencode hebben, of een nog andere opdrachtcode.

Het repertoire van al deze opdrachten voor elementaire handelingen, geschreven in nullen en enen, noemen we de machinetaal. De machine kan alleen opdrachten die in deze machinetaal zijn gegeven uitvoeren. De machinetaal is dus een binaire code en zo zou de opdracht "maak de inhoud van het eerste rekenregister gelijk aan de inhoud van adres 47" in machinetaal (bij een woordlengte van 12 bits) kunnen luiden: 0001 00101111, waarbij 0001 het functiedeel is en 00101111 het adresdeel.

De machinetaal-opdrachten van een programma staan op achtereenvolgende adressen in het geheugen. Zij staan daar als binaire codes en als men het geheugen zou kunnen uitlezen, zou men geen verschil zien tussen de genoemde opdracht en de binaire code van het getal 303. Als de genoemde binaire code in het besturingsorgaan terechtkomt, zal het als de genoemde opdracht worden geïnterpreteerd; komt de code echter in het rekenorgaan terecht, dan zal het als de binaire voorstelling van het getal 303 worden geïnterpreteerd. Als de inhoud van een woord een opdracht is en deze inhoud komt in het rekenorgaan terecht, zal in het algemeen een niet gewilde, foutieve toestand ontstaan.

Twee karakteristieken van de machinetaal zijn dus:
- de lay-out van de opdrachten wordt voorgeschreven door de apparatuur;
- in het geheugen ontbreekt het onderscheid tussen opdrachten en gegevens.

Bij de eerste computers moest een machinetaal-programma, met opdrachten in binaire code, opdracht na opdracht met behulp van schakelaars in het geheugen gebracht worden. Als het programma en de gegevens waarop het programma moest werken dan in het geheugen stonden, kon men met weer andere schakelaars dit programma laten verwerken. Het schrijven van programma's in machinetaal en het verwerken van deze programma's op de zojuist genoemde wijze werkt fouten in de hand en is erg omslachtig. Er zijn een aantal ontwikkelingen te noemen, die het werk van de programmeur hebben vereenvoudigd; één ervan is de ontwikkeling op het gebied van de programmeertalen.

Aan het schrijven van programma's in binaire code kleeft een aantal grote bezwaren:
- De oplossing van het probleem is afhankelijk van de hardware.
- Omdat het effect van een elementaire machinehandeling gering is, worden programma's lang.
- Er treden veel fouten op bij het programmeren, mede door het feit dat er maar twee symbolen zijn: 0 en 1.
- Er moet een hele boekhouding bijgehouden worden van de gebruikte adressen, en het weglaten of tussenvoegen van opdrachten brengt een wijziging in het adresdeel van veel opdrachten met zich mee.
- Uitwisseling van programma's is niet mogelijk, omdat verschillende computers meestal verschillende elementaire handelingen kennen met verschillende representaties voor de opdrachten.

Het grootste probleem is dat de taal is afgestemd op de machine en niet op de programmeur. Om aan deze bezwaren tegemoet te komen is al vrij snel gestreefd naar een andere notatie om programma's in vast te leggen. Er is dan echter in het computersysteem een programma vereist dat een programma in de gebezigde notatie omzet in een programma in machinetaal: *de vertaler*.

1.3 DE EERSTE ONTWIKKELINGEN

De eerste vereenvoudiging die werd ingevoerd, was het schrijven van het opdrachtdeel en het adresdeel in octale, hexadecimale of decimale notatie. Bovendien maakte men programma's, geschreven in binaire code (zo'n programma noemt men het *invoerprogramma*), die programma's geschreven in decimale code konden omzetten in equivalente programma's in binaire code en ervoor zorgden dat deze werden uitgevoerd. Men moest, als het decimaal gecodeerde programma op ponskaarten of ponsband was vastgelegd, aan het invoerprogramma opgeven waar het programma in het geheugen moest worden geplaatst. Voor dit soort complicaties moesten nieuwe opdrachten worden ingevoerd. Deze opdrachten behoorden niet tot de eigenlijke procesbeschrijving. Ze bevatten informatie voor het invoerprogramma en speelden dan ook alleen een rol tijdens de omzettingsfase van decimale naar binaire code en niet meer tijdens de uitvoeringsfase (*pseudo-opdrachten*).

Ook al was het gebruik van een decimale machinetaal een vooruitgang, toch blijven ook hier de eerder genoemde nadelen gelden:
- De wijze waarop de oplossing van een probleem wordt geformuleerd is afhankelijk van de hardware.
- Doordat het effect van de opdrachten zeer beperkt is, worden de programma's relatief lang en is het programmeren tijdrovend.
- Doordat de code nietszeggend is en doordat het programma lang wordt, is er een grote kans op het maken van fouten. Zo kan men bijvoorbeeld bij sprongopdrachten de adressen vaak pas later invullen en dit leidt al vlug tot fouten.
- Wijzigingen in een programma zijn moeilijk aan te brengen. Als er bijvoorbeeld een opdracht moet worden tussengevoegd, moet er rekening mee gehouden worden dat alle volgende opdrachten een ander adres krijgen; dit heeft weer consequenties voor andere opdrachten, zoals sprongopdrachten, die aan deze adressen refereren.
- De code is machinegebonden en daardoor is uitwisseling van programma's niet mogelijk.

Aan deze nadelen komen de *assembleertalen* (symbolische talen) voor een deel tegemoet. De ontwikkeling van deze talen is omstreeks 1952 ingezet.

In de assembleertalen wordt de opdrachtcode uit de opdracht geschreven met een letteraanduiding, die de betekenis enigszins uitdrukt. Zo zou de opdracht "Zet de inhoud van het woord met adres n in rekenregister A" kunnen worden beschreven als: HPAI n (Haal Positief in register A de Inhoud van adres n). We noemen de code HPAI een mnemotechnische code. Aldus wordt het gemakkelijker de codes voor de opdrachten te onthouden, en een programma wordt ook beter leesbaar.

Naast de genoemde vereenvoudiging in de opdrachtcode hebben

de assembleertalen nog de karakteristiek dat met symbolische adressen kan worden gewerkt. Men kan een adres een naam geven en van deze naam gebruikmaken in de opdrachten. Deze mogelijkheid vermindert de kans op schrijffouten en vergissingen aanzienlijk.

Het programma, geschreven in de assembleertaal, moet worden omgezet in een programma in binaire machinetaal. Dit wordt verzorgd door een vast vertaalprogramma, het *assembleerprogramma*. Het assembleerprogramma zet de mnemotechnische codes om in binaire codes en kent absolute adressen toe aan de symbolische adressen. Ook nu worden opdrachten in het programma opgenomen, die alleen dienen om het assembleerprogramma de benodigde informatie te verschaffen: de pseudo-opdrachten.

De assembleertaal staat zeer dicht bij de machinetaal: iedere opdracht in het assembleertaal-programma komt overeen met één opdracht in een machinetaal-programma. Van de programmeur wordt nog steeds een relatief grote kennis van de computer verwacht en de taal is nog steeds machine-afhankelijk doordat het repertoire van opdrachten door de machine wordt voorgeschreven.

Bovendien is een programma nog steeds relatief lang. Om aan dit laatste bezwaar tegemoet te komen zijn vanaf ongeveer 1960 de assembleertalen uitgebreid met het zogenaamde *macromechanisme*. Deze faciliteit komt erop neer, dat aan een rij opdrachten, die tezamen een bepaalde actie bewerkstelligen, een naam wordt verbonden (macrodefinitie) en dat de programmeur in plaats van die rij opdrachten steeds de naam kan noemen (macro-instructie) als de beoogde actie uitgevoerd moet worden. Deze faciliteit lijkt wel wat op het proceduremechanisme; het grote verschil tussen de procedure en de macro is echter, dat de procedure-aanroep tijdens executie tot gevolg heeft dat het desbetreffende deelproces wordt uitgevoerd, terwijl de macro-instructie tijdens de vertaalfase wordt vervangen door de beschrijving van het deelproces (de macrodefinitie). Daarna worden de voor de macro-opdracht ingevulde opdrachten normaal vertaald, samen met de rest van het programma. Het vervangen van de macro-opdracht door de bijbehorende opdrachten gebeurt door de macroprocessor, die kan worden gezien als een uitbreiding op het assembleerprogramma. Het macromechanisme is ook in veel hogere programmeertalen aanwezig.

We hebben nu de ontwikkeling in de machinegebonden talen gezien. De assembleertaal met macrodefinitiemogelijkheid is reeds een grote verbetering ten opzichte van de binaire code. De assembleertaal staat echter nog steeds dicht bij de machine.

In de volgende paragrafen zullen we de ontwikkeling van de hogere programmeertalen bekijken, die dichter bij de mens en bij de op te lossen problemen staan. Daar deze talen echter steeds verder van de machine vervreemden, krijgen we er wel een probleem bij, namelijk dat van de vertaling van programma's in deze hogere talen naar equivalente programma's in machinetaal. We spreken van een

bronprogramma dat moet worden vertaald naar een *doelprogramma*. Dit geldt uiteraard ook al voor de uitgebreide assembleertaal.

Het werken met de computer is aanzienlijk gemakkelijker geworden door het scheppen van (hogere) programmeertalen, waarvan de elementaire statements 'machtiger' zijn dan die van de machinetaal.

1.4 VERGELIJKING HOGERE TALEN EN MACHINETAAL

Een machinetaal is gebonden aan een bepaald type machine; de hogere programmeertaal is machine-onafhankelijk gedefinieerd. Dit levert voor beide typen talen voor- en nadelen op. We zullen hier de voor- en nadelen van de hogere programmeertalen bespreken.

Voordelen

- Een hogere programmeertaal sluit min of meer aan bij de wijze van formuleren en de wijze van denken in het vakgebied waarvoor de taal is ontworpen. Hierdoor is de taal eenvoudig te leren en is de gebruiker uit het vakgebied gemakkelijker in staat zelf direct verwerkbare programma's te schrijven. Dit geldt echter niet voor alle hogere programmeertalen.
- De programmatekst in een hogere programmeertaal is over het algemeen goed leesbaar. Hierdoor is deze tekst op zich al een goed stuk documentatie, kunnen fouten beter worden gelokaliseerd, en is verbeteren van fouten relatief eenvoudig.
- In een programma in machinetaal legt de programmeur zelf geheugen- en registerplaatsen vast, hetgeen een behoorlijke kennis van de machine vereist. Voor een hogere programmeertaal is dit niet nodig.
- De opdrachten (statements) in de hogere programmeertaal zijn machtiger en de mogelijkheden tot het definiëren van datastructuren uitgebreider.
- Programma's geschreven in machinetaal zijn niet overdraagbaar naar machines van een ander type. Door hun machine-onafhankelijke definitie is dit voor hogere programmeertalen wel het geval. De eerlijkheid gebiedt ons hierbij aan te tekenen, dat deze machine-onafhankelijkheid niet altijd even goed is gerealiseerd.
- Door de al eerder genoemde punten is een hogere programmeertaal geschikt om algoritmen in vast te leggen en te publiceren, waardoor uitwisseling van algoritmen mogelijk is.
- De vertaler kan het programma controleren op fouten.

Nadelen

- De vertaling die een programma in de hogere programmeertaal moet ondergaan kost tijd: de vertaaltijd; een machinetaalprogramma vraagt geen of minder vertaaltijd. Deze vertaling kost ook

extra geheugenruimte. Het vertaalprogramma wordt *compiler* genoemd.
- Bij een programma in een hogere programmeertaal wordt een doeltaalprogramma gegenereerd, dat ten aanzien van geheugenbezetting en executietijd niet optimaal behoeft te zijn. Gedeeltelijk is dit te verbeteren door in een programma gebruik te maken van in het systeem - en bij voorkeur in doeltaal - gedeclareerde deelprocessen. Bovendien genereren tegenwoordige vertalers een redelijk efficiënt doeltaalprogramma.
- Sommige hogere programmeertalen hebben zo'n veelvoud aan constructiemogelijkheden, dat deze talen moeilijk te leren en (optimaal) te gebruiken zijn.

1.5 DEFINITIE VAN EEN PROGRAMMEERTAAL

Een programmeertaal moet vastgelegd, gedefinieerd, worden. Deze definitie van de taal is terug te vinden in de vertaler. Men gaat soms zo ver dat men zegt dat de taal wordt gedefinieerd door de vertaler. De vertaler zorgt ervoor dat de computer, die alleen machinetaal-programma's kan verwerken, zich gedraagt als een machine die programma's, geschreven in de bij die vertaler behorende programmeertaal, kan verwerken. In de definitie van de taal worden de mogelijkheden van de taal vastgelegd. De definitie moet antwoord geven op een aantal belangrijke vragen, zoals:
- Wat is het alfabet van basissymbolen waaruit gekozen kan worden bij de notatie van het programma?
- Welke typen waarden zijn in de taal aanwezig; alleen enkelvoudige waarden of ook ingewikkelder structuren? Hoe worden deze structuren vastgelegd?
- Welke operaties kunnen worden uitgevoerd op ieder van de datastructuren en wat is het effect van ieder van deze operaties?
- Welke elementaire opdrachten kunnen we in een programma gebruiken en wat is het effect van ieder van deze opdrachten?
- Wat is de structuur van een programma in deze taal? Is deze structuur overzichtelijk en een afspiegeling van het oplossingsproces voor ons probleem?

Voor de gebruiker die een bepaalde taal wil kiezen, zijn ook nog andere vragen van belang, zoals:
- Wordt de taal veel gebruikt, zodat gebruik kan worden gemaakt van al bestaande programma's?
- Is de taal afgestemd op het toepassingsgebied?
- Is er een vertaler aanwezig en hoe efficiënt is deze? Levert deze vertaler goede foutmeldingen?
- Hoe moeilijk is het om de taal te leren en zijn er geen andere, eenvoudiger talen voor hetzelfde toepassingsgebied?

De definitie van een taal bestaat uit:
- De *syntaxis* (grammatica), die vastlegt welke basissymbolen ter beschikking staan en welke constructies en elementaire opdrachten eruit kunnen worden samengesteld.
- De *semantiek*, die vastlegt wat de betrekkingen tussen de toegelaten constructies zijn en die aan de constructies een betekenis geeft.

Daarnaast kent een taal ook nog een *pragmatisch* aspect, waarbij het gaat om de interpretatie van de constructies door mens of machine.

In hoofdstuk 3 zullen we zien hoe de syntaxis van een programmeertaal formeel kan worden gedefinieerd. Niet van alle programmeertalen is de syntaxis formeel vastgelegd hetgeen onduidelijkheden in de hand kan werken. Ook de semantiek zou formeel beschreven moeten kunnen worden; daaraan wordt gewerkt, maar tot nu toe gebeurt het vaak informeel in een natuurlijke taal.

In hoofdstuk 4 besteden we aandacht aan de semantiek.

1.6 ONTWIKKELING VAN HOGERE PROGRAMMEERTALEN

1.6.1 Eerste generatie van hogere programmeertalen (1950-1958)

De eerste generatie hogere programmeertalen werd ontwikkeld in een tijd dat de hardware nog duur was en er weinig machines waren. Er was dan ook weinig vertrouwen in de economische haalbaarheid van hogere talen.

Toch was deze periode bijzonder vruchtbaar. Veel van de ideeën ten aanzien van hogere talen werden in die tijd geboren en ingebracht in talen als FORTRAN I, ALGOL 58 en IPL 5; de globale richting voor verdere ontwikkelingen werd reeds aangegeven.

1.6.2 Tweede generatie van hogere programmeertalen (1959-1961)

De vier tweedegeneratietalen FORTRAN II, ALGOL 60, COBOL en LISP zijn de eindprodukten van een periode van verhoogde activiteiten met betrekking tot taalontwikkeling en ze vormen een tamelijk stabiele basis voor de verdere ontwikkeling op taalgebied.

FORTRAN

De computer werd in de beginperiode vooral gebruikt voor technisch-wetenschappelijke toepassingen en het is dan ook niet verwonderlijk dat de ontwikkeling van hogere programmeertalen juist in deze richting is begonnen. FORTRAN is één van de eerste hogere pro-

grammeertalen. (De eerste aanwijzen is wat moeilijk, omdat het verschil tussen een uitgebreidere symbolische machinetaal en een hogere taal niet zo duidelijk is.) FORTRAN is ontwikkeld door de computerfabrikant IBM en de taal sluit (of sloot in de beginperiode) nog vrij dicht aan bij de symbolische machinetalen.

Een FORTRAN-programma bestaat uit een hoofdprogramma, een of meer subroutines en zogenaamde data-subprogramma's. De namen van deze subroutines en data-subprogramma's kunnen in het hele programma gebruikt worden (ook in de subroutines en data-subprogramma's); de naam van een subroutine kan echter niet in de subroutine zelf gebruikt worden (geen recursie). Andere namen zijn lokaal ten opzichte van het (sub)programma waarin ze gebruikt worden.

Er bestaat ook een mogelijkheid om globale variabelen te introduceren (COMMON) en om verschillende variabelen met dezelfde geheugenplaats te laten corresponderen.

ALGOL 60

In 1962 verscheen het definitieve rapport van ALGOL 60. Deze taal is ontworpen door een internationale commissie, die de taal heeft vastgelegd in een rapport (The revised Report on the Algorithmic Language ALGOL 60). Dit rapport is uitgebracht onder auspiciën van IFIP (International Federation for Information Processing).

Een van de grootste verdiensten van ALGOL 60 is dat het de eerste uitgebreidere hogere programmeertaal is met een geformaliseerde syntaxis. ALGOL 60 heeft veel bijgedragen tot de theorie van programmeertalen.

De taal is zeer geschikt voor het maken van procesbeschrijvingen. Doordat echter FORTRAN als het ware de oudste rechten heeft, wordt FORTRAN veel meer gebruikt dan ALGOL 60, zeker in de Verenigde Staten.

Een ALGOL-programma bestaat uit een blok. Een blok bestaat uit een serie declaraties, gevolgd door een rij statements. Eén van de vormen van een statement is weer een blok, zodat willekeurig diep 'geneste' blokken kunnen voorkomen. Alle variabelen moeten gedeclareerd zijn in het blok waarin ze gebruikt worden of in een omvattend blok hiervan. Deelprocessen kunnen vastgelegd worden in procedures. Procedures kunnen ook zichzelf activeren (recursie).

COBOL

COBOL is ontworpen voor administratieve toepassingen (werken met bestanden) en is dan ook in zijn mogelijke datastructuren en opdrachten hierop afgestemd; bovendien is veel aandacht besteed aan invoer- en uitvoermogelijkheden omdat in de administratie de opmaak (lay-out) van gegevens belangrijk is. Voordat COBOL beschikbaar was werd voor dit soort werk veel gebruik gemaakt van symbolische machinetaal. Aan rekenkundige opdrachten is minder

aandacht besteed, omdat de bewerkingen in de administratie vrij eenvoudig zijn. De taal lijkt in zijn schrijfwijze wel wat op Engels. COBOL is ontstaan (1960) uit een samenwerking tussen een groot aantal commissies.

Een COBOL-programma bestaat uit een IDENTIFICATION DIVISION waarin de naam van de programmeur en de naam van het programma worden vastgelegd, een ENVIRONMENT DIVISION waarin de hardware wordt vastgelegd en waarin een relatie wordt gelegd tussen de logische en fysieke invoer en uitvoer, een DATA DIVISION waarin de structuur van de gegevensbestanden wordt vastgelegd en een PROCEDURE DIVISION die de eigenlijke procesbeschrijving bevat.

COBOL was de eerste taal waarin de recordstructuur voor data aanwezig was.

LISP

De taal is ontworpen door een groep mensen onder leiding van de wiskundige McCarthy. De taal is bedoeld voor de verwerking van lijsten van gegevens. LISP is gebaseerd op wiskundige grondslagen en is dan ook zeer geformaliseerd. Een programma heeft een eenvoudige structuur, maar doordat de taal slechts een klein arsenaal basiselementen bevat, zijn programma's vaak moeilijk te begrijpen. Zoals ALGOL 60 is ook LISP een belangrijke ontwikkeling geweest in de theorie van de programmeertalen.

1.6.3 Derde generatie van hogere programmeertalen (1962-1969)

PL/I

PL/I is een (niet zo geslaagde) poging om alle goede eigenschappen van de verschillende talen in één taal onder te brengen. Het is een produkt van de computerfabrikant IBM. Veel aandacht is besteed aan de definitie van de semantiek.

ALGOL 68

Ook ALGOL 68 is een taal met zeer veel mogelijkheden. Dit wordt bereikt door een (beperkt) aantal concepten te introduceren die op willekeurige wijze met elkaar gecombineerd kunnen worden. Het is een blok-gestructureerde taal met uitgebreide mogelijkheden om nieuwe typen te introduceren. Ook is het mogelijk parallelle processen te creëren.

SNOBOL4

SNOBOL is een taal waarvan de ontwikkeling in 1962 is gestart op de Bell Telephone Laboratories in de Verenigde Staten. De taal is gebaseerd op de ideeën van Markov-algoritmen. Een programma

bestaat in principe uit een aantal transformatieregels. In 1967 verscheen de definitieve versie (SNOBOL4).

De taal vindt vooral toepassing in programma's waarin met teksten moet worden gewerkt.

SIMULA 67

SIMULA introduceerde een nieuwe programma-eenheid: de class. Hierin zijn datatypen en operaties opgenomen. Het bijzondere ervan is, dat als de class eenmaal is geactiveerd, deze blijft bestaan.

Het class-concept heeft grote invloed (gehad) op de ontwikkeling op het terrein van datastructuren. We komen hierop terug in hoofdstuk 7.

APL

APL kent een groot aantal operatoren die op arrays werken. APL kent geen declaraties en heeft slechts een beperkt aantal besturingsstructuren. De taal geeft aanleiding tot een manier van programmeren, die de correctheid en de onderhoudbaarheid van programma's niet ten goede komt. De taal (en het ondersteunende systeem) wordt interactief gebruikt, vooral voor technisch/wetenschappelijke toepassingen.

BASIC

De taal is bedoeld voor beginnende programmeurs (Beginner's All-purpose Symbolic Instruction Code). De taal wordt interactief gebruikt. Doordat hogere structuren niet aanwezig zijn, werkt de taal 'slecht' programmeren in de hand (bijvoorbeeld doordat overmatig gebruik van de sprongopdracht noodzakelijk is).

1.6.4 Vierde generatie van hogere programmeertalen (1970- ....)

Pascal

Pascal is een ALGOL-achtige taal die door N. Wirth is ontworpen. De taal beschikt over een groot aantal mogelijkheden voor datastructurering. Het doel van het ontwerp was een eenvoudige taal te creëren waarin programma's geschreven konden worden die efficiënt vertaald zouden kunnen worden. Bovendien was de betrouwbaarheid van programma's erg belangrijk.

Veel nieuwe ontwikkelingen zijn gebaseerd op Pascal.

Ada

De nieuwe programmeertaal Ada is ontworpen in opdracht van het ministerie van defensie van de Verenigde Staten. In het rapport wordt het toepassingsgebied omschreven als "embedded computer

systems". De taal is gebaseerd op Pascal, maar kent uitgebreidere mogelijkheden voor datastructurering en heeft ook constructies voor parallelle programmering.

2 TAAL

2.1 INLEIDING

In de vorige hoofdstukken is steeds gesproken over programmeer-*talen*. Is de notatie waarin we algoritmen vastleggen wel een taal? Om deze vraag te kunnen beantwoorden moeten we weten wat een taal is. Zolang iemand slechts één taal spreekt en schrijft, is "wat is een taal?" geen echte vraag voor hem. Taal is wat hij schrijft en spreekt. Als we alleen natuurlijke talen beschouwen, kan het begrip taal verduidelijkt worden door voorbeelden van talen te noemen, zoals Nederlands, Engels en Duits. Maar zodra kunstmatige of geconstrueerde talen een rol spelen, is het niet zo duidelijk waar de grens voor het begrip ligt. Is machinetaal werkelijk een taal of is het slechts een code? Maar wat is het verschil tussen een taal en een code? Het feit dat de vastgelegde informatie hetzelfde kan zijn, of het nu beschreven is in Nederlands, ALGOL of machinetaal, geeft nog geen enkel houvast bij de beantwoording van de vraag die we ons stelden. Het is wel duidelijk dat sommige typen kunstmatige talen veel dichter bij natuurlijke talen staan dan andere typen. Maar de algemene theorie van talen wil alle vormen van taal omvatten. We kunnen dus onmogelijk primitieve systemen uitsluiten. We kunnen natuurlijk wel proberen een classificatie aan te brengen.

Het woord 'taal' schijnt dezelfde afkomst te hebben als het woord 'getal'. En taal zou dan ook afkomstig kunnen zijn van het uitspreken of opschrijven van getallen. Met het woord 'language' ligt het eenvoudiger, omdat dit woord samenhangt met 'tongue' en dus zou slaan op gesproken tekst.

Naast het feit dat een programmeertaal een stuk ontwerpgereedschap is voor algoritmen, is het ook een communicatiemiddel tussen mens en machine en tussen mensen onderling. In een programmeertaal kunnen programma's geformuleerd worden, die door een machine uitgevoerd worden.

Met behulp van een programmeertaal kunnen we met anderen communiceren over de oplossingen die we voor programmeerproblemen hebben gevonden. Zoals iedere taal is ook de programmeertaal de drager en de representatie van ideeën: daar het erg

moeilijk is ideeën in abstracte vorm te hanteren, is de taal een belangrijk instrument voor het uitdrukken, verfijnen en preciseren van ideeën. Een programmeertaal is dus tevens een communicatiemiddel voor de programmeur met zichzelf. Dit aspect treedt ook op bij de aanpassing en bewerking van programma's; het is nu eenmaal zo dat slechts in uitzonderingsgevallen de eerste versie van een programma correct is en het uiteindelijke programma vormt.

2.2 TAALTHEORIE

In de theorie van talen, ook wel *semiotiek* genoemd, onderscheidt men drie gebieden van onderzoek, te weten: *syntaxis*, *semantiek* en *pragmatiek*.

Een taal, en dus ook een programmeertaal, is een systeem van karakterrijen waarmee woorden, uitdrukkingen, zinnen en omvangrijke samenvoegingen gevormd kunnen worden. De karakters komen uit een bepaalde verzameling die we *alfabet* noemen. De syntaxis handelt over de combinatie van karakters tot karakterrijen zonder dat er gelet wordt op hun specifieke betekenis of op de relatie tot de context waarin ze voorkomen. Kort gezegd is de syntaxis de leer van de samenvoeging van karakters. De semantiek handelt over de betekenis van karakters en karakterrijen, het is de leer van de relatie tussen de karakters en karakterrijen en de objecten waarop ze van toepassing zijn. De pragmatiek handelt over het gebruik en het effect (binnen de context waarin ze gebruikt worden) van karakters en karakterrijen. De pragmatiek is de leer van de relatie tussen karakters en interpretatoren.

Door het invoeren van de drie begrippen syntaxis, semantiek en pragmatiek wordt gesuggereerd dat er een scherpe scheiding is tussen deze gebieden. Dit is echter geenszins het geval, de drie dimensies van de semiotiek zijn eerder beschouwingswijzen over en houdingen ten opzichte van een taal dan absolute eigenschappen hiervan. De grenzen tussen de drie gebieden zijn vaag en kunnen verplaatst worden door op een andere wijze de taal te beschouwen.

De syntactische regels om te combineren, karakters tot woorden, woorden tot zinnen en willekeurige elementen tot ingewikkelde patronen, zijn in het ideale geval duidelijk en zonder uitzonderingen te formuleren. Dan kan van een (formele) tekst met behulp van een automaat nagegaan worden of deze tekst syntactisch correct is. Gegeven de regels en bepaalde selectiecriteria (die ingegeven kunnen zijn door een hoger niveau – zoals dat van de betekenis – of louter berusten op een random trekking) kunnen door automatische mechanismen formele teksten gegenereerd worden en kunnen formele teksten onderverdeeld (ontleed) worden in de syntactische componenten. We komen hier in het volgende hoofdstuk op terug.

De betekenis van een constructie in een taal heeft te maken met de relatie tussen die tekst en de wereld die beschreven wordt, de object-wereld. Het nagaan of de betekenis, die de tekst heeft, overeenkomt met de beschreven wereld, is over het algemeen een groot probleem. In het ideale geval, dat alleen bij een geconstrueerde taal kan optreden, kunnen er regels zijn die aangeven hoe de betekenis van een tekst kan worden vastgesteld. Hierbij kunnen we alleen spreken over de objectieve betekenis, daar de regels iedere subjectiviteit uitsluiten. Er is altijd semantiek, tenzij we een zinloos spelletje met karakters spelen; in dat laatste geval kunnen we volstaan met de syntaxis. We kunnen de semantiek beschouwen en zelfs formaliseren onafhankelijk van de precieze notatie van de constructies, zodat de semantiek is geïsoleerd van de syntaxis. Voor het vastleggen van de syntaxis en de semantiek moet weer gebruik gemaakt worden van een taal, de *metataal* (zie hieronder).

Om syntaxis en semantiek te onderscheiden moeten we vorm en betekenis scheiden. Door de wiskundige (formele) aard van de beschrijving van de syntaxis is dit een moeilijke zaak. Nergens anders dan juist in de wiskunde zijn vorm en betekenis in zo'n grote mate verstrengeld. Formalisering is juist het verpakken van zo veel mogelijk betekenis in gedefinieerde vormen. Tot op zekere hoogte wordt dit ook in natuurlijke talen gedaan. Het meervoud is bijvoorbeeld een syntactische eenheid met een duidelijk herkenbare vorm en een eigen betekenis. In geconstrueerde talen wordt dit ook gedaan, maar dan veel verder doorgevoerd.

De syntactische elementen van geconstrueerde talen vertegenwoordigen al een belangrijk stuk betekenis. Pragmatisch is dit erg belangrijk, omdat de mens tijdens het lezen een soort betekenistoekennend mechanisme heeft dat aanleiding kan geven tot fouten als de kunstmatig toegekende betekenis afwijkt van de gewoonlijke, natuurlijke betekenis. Als een aritmetische expressie in zijn notatie lijkt op de ons bekende rekenkundige uitdrukking, maar bij verwerking door de machine op een andere wijze dan wij gewoon zijn wordt geïnterpreteerd, zal dat ongetwijfeld tot fouten leiden. Het pragmatische aspect is altijd aanwezig, daar een taal zonder gebruiker geen enkele zin heeft. Voor geconstrueerde talen is het verleidelijk te gaan spelen met syntactische regeltjes, maar eigenlijk moet er juist vanuit de pragmatiek gestart worden: welke functies zijn nodig, welke doeleinden moeten bereikt worden, wat is een aantrekkelijke notatie voor een bepaalde abstracte constructie? Zowel bij de natuurlijke talen als bij de geconstrueerde talen is de pragmatiek dat deel van taalonderzoek, dat het minst geschikt is voor een formele behandeling. Ook de scheiding tussen semantiek en pragmatiek is moeilijk precies aan te geven.

Als we een taal gaan beschrijven of iets over een taal willen zeggen, hebben we een taal nodig. Zo'n taal wordt een *metataal* genoemd. Bij beschouwingen over talen spelen drie niveaus een rol: de beschre-

ven wereld (object-wereld), de taal waarmee deze wereld wordt beschreven (object-taal) en de taal waarin de taal wordt beschreven (metataal). Wat de metataal betreft kunnen we weer over de drie beschouwingswijzen (syntaxis, semantiek en pragmatiek) spreken. Zo kunnen we ons willekeurig veel niveaus van beschouwingen voorstellen en soms wordt dit ook gedaan, maar op een gegeven moment zal er altijd een stap gedaan moeten worden naar de uiteindelijke metataal: de natuurlijke taal.

2.3 FORMELE TALEN; GECONSTRUEERDE TALEN

De begrippen 'formeel' en 'geconstrueerd' zijn niet identiek. Er zijn talen waarvoor slechts één van deze karakteriseringen van toepassing is: er zijn informele kunstmatige talen en ook een natuurlijke taal kan (voor een deel) geformaliseerd worden.

Vaak worden formalisatie en syntaxis als een en hetzelfde begrip beschouwd. Syntactische regels kunnen echter zeer informeel zijn, ze zijn dit ook eeuwen geweest. En het is zeker mogelijk ook semantiek en pragmatiek te formaliseren. Vooral voor programmeertalen is het belangrijk dat alle drie de semiotische dimensies van deze talen geformaliseerd worden, zodat de inhoud en het gebruik van de taal precies zijn vastgelegd.

Een programmeertaal dient een geformaliseerde, geconstrueerde taal te zijn. De syntaxis en semantiek van programmeertalen dienen met de grootst mogelijke nauwkeurigheid beschreven te worden om er zeker van te zijn dat de machine precies datgene realiseert wat de programmeur bedoelt. Andere redenen voor de formalisering zijn:
- impliciete veronderstellingen worden uitgesloten;
- over de begrippen mag geen misverstand bestaan;
- constructies kunnen een vaste betekenis krijgen;

Redenen voor de toepassing van kunstmatige talen zijn:
- de syntaxis en semantiek van natuurlijke talen zijn niet volledig formeel te geven;
- er kan een eigen begrippenarsenaal ingevoerd worden, los van de te beschrijven wereld;
- begrippen kunnen zo ingevoerd worden dat ze eenduidig zijn;
- de notatie kan zo gekozen worden dat met weinig woorden veel 'gezegd' kan worden.

Het kan zijn dat de uiteindelijke metataal een natuurlijke taal moet zijn; dit hoeft geen reden te zijn om ook voor de eerste metataal of voor de programmeertaal een natuurlijke taal te kiezen. Er wordt nogal eens gepleit voor het gebruik van natuurlijke talen bij de programmering. De voornaamste argumenten die hiervoor worden aangevoerd zijn de grote algemeenheid en de gemakkelijke beschikbaarheid (welhaast iedereen spreekt en schrijft een taal) van de

natuurlijke talen. Bij de programmering hebben we echter geen behoefte aan de grote algemeenheid, en de gemakkelijke beschikbaarheid zou ons kunnen doen vergeten dat we bij de programmering een brug slaan tussen een reële en een geconstrueerde wereld.

Een argument dat vaak gehanteerd wordt tegen een formele notatie is dat de teksten zoveel moeilijker te lezen, zoveel mystieker worden. Het kost natuurlijk enige moeite om een formele notatie te leren, maar de winst die we kunnen halen komt tot uitdrukking in vele facetten: duidelijkheid, eenduidigheid, betrouwbaarheid en kortere teksten. Bovendien is de tekst in een geconstrueerde taal onafhankelijk van de natuurlijke moedertaal van de lezer en de schrijver. Een beschrijving in een natuurlijke taal is nooit helemaal precies, volledig of vrij van tegenspraak. De termen kunnen bijbetekenissen of associaties oproepen die buiten de bedoeling van de schrijver liggen. Een formele, geconstrueerde taal kan ook beter de gelegenheid scheppen tot structurering en abstractie.

Bij het gebruik van een programmeertaal moet uitgegaan kunnen worden van de (exact beschreven) semantiek van de te gebruiken constructie om dan later op zoek te gaan naar de juiste syntactische eenheden voor de beschrijving (eerst weten *wat*, daarna *hoe*). Het komt al te vaak voor dat syntactische vormen worden gekozen die later moeilijkheden opleveren bij het realiseren van de gewenste semantiek. Deze wijze van werken treedt vooral op als men niet beschikt over formele definities. De beoordeling van een programmeertaal gebeurt – ten onrechte – vaak alleen ten aanzien van de syntaxis.

Het gebruik van een formele syntaxis is reeds ingeburgerd. De standaardmethode is het toepassen van voortbrengingsregels waarmee correcte (well-formed) zinnen uit de taal kunnen worden gegenereerd. Een compleet systeem van deze regels wordt een generatieve grammatica genoemd. Deze definitiewijze voor programmeertalen staat bekend als de Backus Normal Form (BNF), omdat de Amerikaan J. Backus een formele definitie van de syntaxis van ALGOL 60 heeft gegeven in de vorm van voortbrengingsregels in een speciale notatie.

Op grammatica's en de relatie tussen talen en interpretatoren zullen we in het volgende hoofdstuk ingaan.

Formele definitie van de semantiek is ingewikkelder. We kunnen trachten het probleem op te lossen door van de constructies een vertaling te geven in een taal waarvan de betekenis bekend verondersteld wordt. Dit komt neer op een verplaatsing van het probleem. In hoofdstuk 4 zullen we aandacht besteden aan de semantiek.

We moeten ons er wel van bewust zijn dat de duidelijke ja-nee beslissing een voorrecht is van de kunstmatige, geconstrueerde wereld van logica en formele definitie.

De computer is een geconstrueerd instrument, zowel in hardware als in software, dat communiceert met de reële, levende wereld. Bij het

afstemmingsprobleem tussen het geconstrueerde instrument en de reële wereld is het behoud van betekenis van het grootste belang. Het goed functioneren van de automaat hangt niet alleen af van betrouwbare hardware en software, het hangt ook af van de betrouwbaarheid van de afbeelding van de reële wereld, van het model van deze wereld waarmee gemanipuleerd gaat worden. De mens heeft een ongelooflijke leercapaciteit en kan, in tegenstelling tot de machine, kijken wat de betekenis is van hetgeen hij doet. De machine volgt alleen de regels op die volgen uit het model. Wat dus in het bijzonder de aandacht verdient is de overgang van natuurlijke naar geconstrueerde informatie, het punt waar de relatie ligt tussen de formele structuur en de levende omgeving. Dit punt ligt bij de modelvorming. De afstand tussen natuurlijke en geconstrueerde grootheden dient zorgvuldig in ogenschouw te worden genomen, het behoud van de betekenis in de informatieverwerking zal steeds precisie vragen in het gebruik van de terminologie. De talen die in en rond de computer gebruikt worden spelen dan ook een belangrijke rol.

Steeds meer mensen zonder een informatica-opleiding zullen gebruik willen en moeten maken van informatieverwerkende faciliteiten. Zij kunnen geen extreem formele systemen gaan leren. Sommige mensen concluderen hieruit dat het programmeren in een natuurlijke taal de oplossing moet zijn. Zoals al eerder gezegd moet men zich realiseren dat er een afstand blijft tussen de reële wereld en de geconstrueerde machine. Het is onnatuurlijke geconstrueerde relaties en programma's in een natuurlijke taal te beschrijven zoals het ook onmogelijk is de reële wereld in een formeel systeem te beschrijven.

2.4 DE PRAGMATIEK

Daar programmeertalen een communicatiemiddel zijn tussen mens en machine, heeft de taal twee soorten gebruikers: natuurlijke en gemechaniseerde gebruikers. De eerste soort gebruiker, de mens, heeft meningen en kan onlogisch zijn; de tweede gebruiker, de machine, is volledig algoritmisch en voert precies uit wat als betekenis in de tekst is vastgelegd. Er zijn dan ook twee soorten aspecten in de pragmatiek, de menselijke aspecten en de mechanische aspecten.

Tot de mechanische pragmatiek behoort het vertaalproces en zeker de efficiëntie-overwegingen die hierbij een rol spelen. Punten die van belang zijn: Wat is de relatie tussen de compiler en de taal? Hoe beïnvloedt een vertaalmethode een taal? Hoe beïnvloeden de compilers de machinetaal-programma's en de resultaten? Wat is de relatie tussen het operationele systeem en de taal? Moeten operating system en taal op elkaar afgestemd zijn en hoe zeer beïnvloeden ze

elkaar? Daarnaast zijn er problemen die samenhangen met de afhankelijkheid van de taal ten opzichte van de machine. En natuurlijk omgekeerd. In het ideale geval zou het totale systeem – programmeertaal, machinetaal, hardware – tegelijkertijd ontworpen moeten worden.

De relatie tussen de programmeertaal en de mens is het laatste maar zeker niet het onbelangrijkste terrein van de pragmatiek. Het kunnen lezen, leren en onderwijzen van programmeertalen zijn psychologische problemen waarvan het succes van een programmeertaal veel meer kan afhangen dan van al zijn technische mogelijkheden.

Een taal is een sociaal gereedschap; ook dat moet in ogenschouw worden genomen als we willen dat een programmeertaal werkelijk een taal wordt. Want het beste logische systeem is geen taal zolang het op het bureau van zijn uitvinder ligt. Een echte taal wordt gekarakteriseerd door het praktische gebruik ervan binnen een bepaalde groep mensen. De taal zal zich hiervoor moeten lenen.

Mens-machine-communicatie is een van de grote ontwikkelingen die ons te wachten staan. De taal zal daaraan aangepast moeten zijn. Dit houdt onder andere in dat de taal eenvoudig zal moeten zijn.

Ook tot de pragmatiek behoort de relatie tussen programmeertaal en toepassingsgebeiden. Welke constructiemogelijkheden dienen in een programmeertaal aanwezig te zijn om deze aantrekkelijk te maken voor een bepaald toepassingsgebied?

2.5 AFSLUITING

Zowel in de taaltheorie als in de programmeerpraktijk is het duidelijk dat talen zowel beschrijvend als voorschrijvend zijn; declaratie en opdracht, toestanden en acties.

Het onderscheid tussen beschrijving en voorschrift is echter niet essentieel. Iedere opdracht is tegelijkertijd de beschrijving van de gewenste toestand en iedere beschrijving kan worden opgevat als een leiddraad voor de constructie. Het tijdsaspect bij de actie is niet belangrijk voor beschouwingen over taal.

3 SYNTAXIS, FORMELE TALEN EN ABSTRACTE AUTOMATEN

3.1 INLEIDING

In hoofdstuk 2 is al gezegd dat ALGOL 60 een belangrijke bijdrage heeft geleverd aan de theorie van programmeertalen; dit geldt met name voor de definitie van programmeertalen. Het ALGOL-rapport introduceerde een nieuwe wijze van definitie van de syntaxis van de taal, die daarna ook voor allerlei andere talen is gebruikt. Deze definitiewijze staat bekend onder een aantal namen: de BNF-notatie, Backus Normal Form (J. Backus was de uitvinder van de notatie), Backus Naur Form (P. Naur was de redacteur van het rapport) en context-vrije grammatica. Deze laatste naam is afkomstig van N. Chomsky, die natuurlijke talen bestudeerde.

Een programmeertaal (en ook een natuurlijke taal) kan geleerd worden door naar veel voorbeelden te kijken en aldus een intuïtief idee te krijgen. Dit is vaak voldoende, maar hiermee kan geen exact antwoord gegeven worden op de vraag: "Is deze constructie in de taal toegestaan?". Deze vraag kan beantwoord worden aan de hand van de syntaxis. Je zou kunnen proberen de programmeertaal, zeg Pascal, te definiëren met behulp van een natuurlijke taal, bijvoorbeeld Nederlands. De taal die gedefinieerd wordt noemt men wel de objecttaal en de taal waarin de definitie wordt gegeven de metataal. In het voorbeeld zou Nederlands de metataal zijn voor de objecttaal Pascal. De natuurlijke talen zijn niet zo geschikt als metataal vanwege de mogelijkheden tot dubbelzinnigheid.

Men zegt ook wel dat de taal wordt gedefinieerd door een compiler, dus waarom nog een andere definitie? Hiertegen zijn enkele bezwaren in te brengen. Ten eerste moet die compiler ook gemaakt worden en waar gaat men daarbij dan vanuit? Ten tweede is het maar de vraag of twee compilers voor dezelfde taal volledig overeenstemmen. En ten derde is een compiler min of meer machine-afhankelijk.

Er zijn twee partijen geïnteresseerd in de definitie van een programmeertaal: de gebruiker van de taal (de programmeur) en de compiler-bouwer. Soms zijn de eisen die deze partijen stellen aan de definitie van een programmeertaal met elkaar in conflict en men gaat

er dan wel toe over om twee (equivalente) definities te geven. Dit geldt met name voor de semantiek (zie volgend hoofdstuk). De gebruiker en de compiler-bouwer zijn er beide in geïnteresseerd dat de compiler correct is, vandaar dat bij de definitie van een taal vaak het belang van de compiler-bouwer prevaleert.

Voor veel talen, ook programmeertalen, geldt dat grote gedeelten van de syntaxis beschreven kunnen worden door grammatica's, die context-vrij of regulier zijn. Een grammatica is een generatief systeem; dat wil zeggen dat iedere constructie uit de desbetreffende taal gevonden kan worden door, uitgaande van een startsymbool, een aantal substitutieregels toe te passen. Naast deze *generatoren* voor talen kunnen we ook *acceptoren* beschouwen. Een acceptor is een abstracte automaat die, als hij een rij symbolen als invoer toegediend krijgt, na eindige tijd stopt en de uitvoer 'ja' geeft als de rij een constructie uit de taal is.

3.2 ENKELE DEFINITIES MET BETREKKING TOT GRAMMATICA'S

Basisterminologie

Een *alfabet* (vocabulaire) is een eindige, niet-lege verzameling; de elementen noemen we *symbolen*. Een eindige rij van symbolen noemen we een *string*; een speciale string is de rij bestaande uit nul symbolen: de *lege string* ε.

De verzameling van alle strings bestaande uit symbolen van alfabet A, inclusief de lege string, wordt aangegeven met A^*. Willen we de lege string uitsluiten dan gebruiken we A^+ ($A^+ = A^* \setminus \{\varepsilon\}$).

De lengte van de string α, aangegeven door $|\alpha|$ of $\ell(\alpha)$, is het aantal symbolen in α; als $\alpha = a_1 a_2 \ldots a_n$ ($a_i \in A$) dan geldt $|\alpha| = n$, als $\alpha = \varepsilon$ dan $|\alpha| = 0$.

Als W_k de verzameling is van alle strings ter lengte k over A en $W_0 = \{\varepsilon\}$, dan geldt:

$$A^* = \bigcup_{k=0}^{\infty} W_k$$

Als $\alpha = a_1 a_2 \ldots a_n$ en $\beta = b_1 b_2 \ldots b_m$ ($a_i \in A$, $b_j \in A$) dan is de *concatenatie*, $\alpha.\beta$, van α en β gelijk aan $a_1 a_2 \ldots a_n b_1 \ldots b_m$. De concatenatie wordt ook wel aangegeven met $\alpha\beta$. De concatenatie $V_1 V_2$ van de verzamelingen V_1 en V_2 is gedefinieerd als $\{xy \mid x \in V_1,\ y \in V_2\}$. Voor de concatenatie van strings geldt:

- $\alpha\beta\gamma = \alpha(\beta\gamma) = (\alpha\beta)\gamma$; de concatenatie is associatief
- $\alpha\varepsilon = \varepsilon\alpha = \alpha$; ε is een eenheidselement voor de concatenatie
- $|\alpha\beta| = |\alpha| + |\beta|$.

Verder gebruiken we de afkorting a^n voor $\overbrace{aa\ldots a}^{n \text{ maal}}$ ($n \geq 0$).
($a^{n+1} = a^n a$, $a^0 = \varepsilon$, $a^1 = a$).

De string α is een *deelstring* van β als er een $\sigma \in A^*$ en een $\tau \in A^*$ bestaan met $\beta = \sigma\alpha\tau$.

In de grammatica (syntaxis) voor een taal L worden twee eindige disjuncte verzamelingen symbolen gebruikt. De ene verzameling, N, is een *hulpalfabet*, waarvan de elementen *hulpsymbolen* (of *non-terminals*) worden genoemd. De andere verzameling, T, is de verzameling van *eindsymbolen* (of *terminals*), het reeds eerder genoemde alfabet. In de grammatica wordt ook gebruik gemaakt van $V = N \cup T$. De taal, die door de grammatica gedefinieerd wordt, bestaat uit strings van symbolen uit T (taal L is gedefinieerd over T; $L \subset T^*$). De grammatica bevat voorts een verzameling produktieregels R. Een element van R is een geordend paar (α, β) met $\alpha \in V^*NV^*$ (α is een string met ten minste één non-terminal) en $\beta \in V^*$. De produktieregels geven aan hoe de constructies uit de taal gegenereerd kunnen worden. In plaats van (α, β) schrijven we meestal $\alpha \rightarrow \beta$. De *taal* L(G), die door de grammatica G gedefinieerd wordt, bestaat uit alle strings over T die met behulp van de produktieregels afgeleid kunnen worden uit een speciaal non-terminal symbool S, het *startsymbool*.

Een *grammatica* G is een geordend viertal (N, T, R, S), waarvoor geldt:
- N is een alfabet, de non-terminals
- T is een alfabet, de terminals ($N \cap T = \emptyset$; $V = N \cup T$)
- R is een eindige verzameling geordende paren (α, β) met $\alpha \in V^*NV^*$ en $\beta \in V^*$
- S is het startsymbool, $S \in N$.

VOORBEELD 1
$G = (\{S, A, B\}, \{p, q, r, s\}, \{S \rightarrow AB, A \rightarrow p, A \rightarrow r, B \rightarrow q, B \rightarrow s\}, S)$.
De taal die door G gedefinieerd wordt bestaat uit de constructies pq, ps, rq en rs.
(einde voorbeeld 1)

Een *zinsvorm* van G wordt gedefinieerd als:
- S is een zinsvorm
- Als $\alpha\beta\gamma$ een zinsvorm is en $\beta \rightarrow \delta$ is een element van R dan is ook $\alpha\delta\gamma$ een zinsvorm ($\alpha, \beta, \gamma \in V^*$).

Een zinsvorm die geen non-terminals bevat wordt een *zin* genoemd. De *taal* L(G), die voortgebracht wordt door de grammatica G, is de verzameling van alle zinnen die door G worden voortgebracht.

Notatie-afspraken
Voor de verschillende verzamelingen gebruiken we verschillende soorten letters om de elementen te noteren.

T : a, b, c, ..., 0, 1, ...
N : A, B, C, ...
V : U, W, ...
T* : x, y, z, ...
V* : α, β, γ, ...

We definiëren een relatie $\Rightarrow$ (of $\underset{G}{\Rightarrow}$) op de V* van de grammatica G als volgt:
Als $\alpha\beta\gamma$ een string is over V* en $(\beta\rightarrow\delta) \in R$, dan $\alpha\beta\gamma \Rightarrow \alpha\delta\gamma$.
We zeggen dat $\alpha\delta\gamma$ *direct afleidbaar* is uit $\alpha\beta\gamma$ (uit $\alpha\beta\gamma$ is $\alpha\delta\gamma$ in één stap te genereren).

Een rij $\alpha_0, \alpha_1, \ldots \alpha_n$ $(n \geqq 0)$ van (niet noodzakelijk verschillende) strings van V* met $\alpha_0 = \alpha$, $\alpha_n = \beta$ en $\alpha_{i-1} \Rightarrow \alpha_i$ voor $i = 1, 2, \ldots, n$ heet een *afleiding* ter lengte n van β uit α. We kunnen deze afleiding aangeven met $\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n = \beta$.

We definiëren de relatie $\overset{n}{\Rightarrow}$ (of $\overset{n}{\underset{G}{\Rightarrow}}$) tussen twee strings van V* als volgt:

$\alpha \overset{n}{\Rightarrow} \beta$ als er een afleiding van β uit α bestaat ter lengte n.
We zeggen dat α β genereert in n stappen.

We definiëren een relatie $\overset{*}{\Rightarrow}$ (of $\overset{*}{\underset{G}{\Rightarrow}}$) tussen twee strings van V* als volgt:

$\alpha \overset{*}{\Rightarrow} \beta$ als er een $n \geqq 0$ is zodanig dat $\alpha \overset{n}{\Rightarrow} \beta$ ($\alpha \overset{*}{\Rightarrow} \beta$ als er een afleiding van β uit α bestaat).

We kunnen nu de taal L(G), die wordt voortgebracht door de grammatica G = (N, T, R, S), noteren als

$L(G) = \{x \mid x \in T^*,\ S \overset{*}{\Rightarrow} x\}$

VOORBEELD 2
$G = (\{A, S\}, \{0, 1\}, R, S)$ met $R = \{S \rightarrow 0A1,\ 0A \rightarrow 00A1,\ A \rightarrow \varepsilon\}$

$L(G) = \{0^n 1^n \mid n \geqq 1\}$
(einde voorbeeld 2)

VOORBEELD 3
$G = (\{A, B, C\}, \{k, l\}, P, A)$ met $P = \{A \rightarrow kBk,\ B \rightarrow l,\ B \rightarrow kBC,\ Ck \rightarrow lkk,\ Cl \rightarrow lC\}$

$L(G) = \{k^n l^n k^n \mid n \geqq 1\}$
(einde voorbeeld 3)

In voorbeeld 1 is de taal een eindige verzameling van zinnen. In het tweede en derde voorbeeld zijn de talen niet-eindige verzamelingen. Dit wordt veroorzaakt doordat er *recursieve* produktieregels voorkomen in de grammatica. Een recursieve produktieregel is een regel waarbij zowel in het linkerlid als in het rechterlid hetzelfde nonterminal symbool voorkomt. Voorbeeld 4 geeft nog een eenvoudig voorbeeld hiervan.

VOORBEELD 4
G = ({S, A}, {0, 1, 2}, {S→0A, A→2, A→1A}, S)
L(G) = $\{01^n2 \mid n \geq 0\}$
(einde voorbeeld 4)

Een grammatica wordt *monotoon* genoemd als voor alle produktieregels $\alpha \rightarrow \beta$ geldt $|\alpha| \leq |\beta|$.

3.3 ENKELE VOORBEELDEN IN EEN ANDERE NOTATIE

Het onderstaande voorbeeld is afkomstig uit de programmeertaal ALGOL 60. Van deze grammatica geven we alleen de produktieregels. In aansluiting op het definiërende rapport van ALGOL 60 gebruiken we een notatie die afwijkt van de eerder gegeven notatie. In plaats van "→" schrijven we "::=" en in plaats van twee regels α ::= β en α ::= γ schrijven we α ::= $\beta|\gamma$.

<number> ::= <decimal number>|<exponent part>|
<decimal number> <exponent part>
<decimal number> ::= <unsigned integer>|<decimal fraction>|
<unsigned integer> <decimal fraction>
<exponent part> ::= $_{10}$<integer>
<decimal fraction> ::= . <unsigned integer>
<integer> ::= <unsigned integer>|+<unsigned integer>|
–<unsigned integer>
<unsigned integer> ::= <digit>|<unsigned integer> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

De elementen van de taal worden genoemd naar het startsymbool: numbers (in het rapport van ALGOL 60 worden de op deze wijze gedefinieerde elementen getallen zonder teken (unsigned numbers) genoemd).

De hulpsymbolen zijn hier tussen het hakenpaar < en > geplaatst. De hulpsymbolen geven we een naam; deze namen duiden enigszins de semantiek aan. Deze produktieregels zeggen niets over de betekenis van de gebruikte eindsymbolen (0..9, $_{10}$, +, –, .). De bovenstaande definitie van getallen zegt dus niets over de semantiek. Maar we kunnen in dit geval aansluiten bij het intuïtieve begrip van getallen en hun waarden.

De bovenstaande notatie is de *BNF*-notatie.

In plaats van recursieve produktieregels, zoals:
<unsigned integer> ::= <digit>|<unsigned integer> <digit>
gebruikt men ook wel de notatie

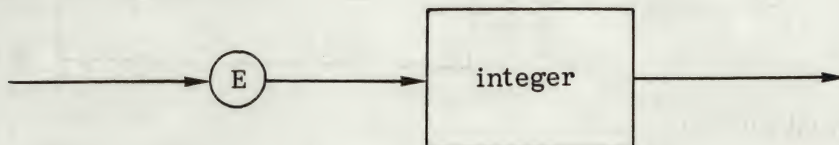
<unsigned integer> ::= <digit>{<digit>}
waarbij {A} aangeeft dat A nul of meer keren kan voorkomen.
In plaats van
<integer> ::= <unsigned integer>|+<unsigned integer>|
-<unsigned integer>
schrijft men ook wel:
<integer> ::= [<sign>]<unsigned integer>
<sign> ::= +|-
Hierbij geeft [A] aan dat A nul of één keer voorkomt.

Men ziet ook wel de notatie $\{A\}_m^n$, die aangeeft dat A minimaal m keer en maximaal n keer voorkomt.

Een andere manier om syntaxregels te beschrijven is met behulp van *syntaxdiagrammen*. De toegestane rijen symbolen van een taal (-constructie) worden beschreven door een diagram met één inkomende pijl en één uitgaande pijl. Elk pad door het diagram, in de richting van de pijlen, komt overeen met een toegestane rij symbolen.

Bijvoorbeeld:

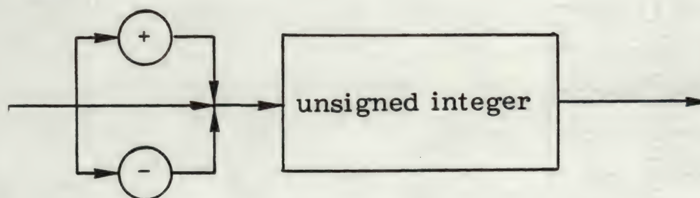
exponent part :



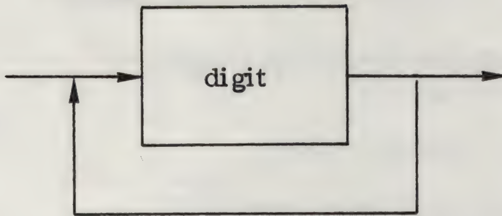
Eindsymbolen staan in cirkels, hulpsymbolen in rechthoekige hokjes. Alternatieven of repetities worden aangegeven met splitsingen of teruggaande pijlen in een syntaxdiagram.

Bijvoorbeeld:

integer :



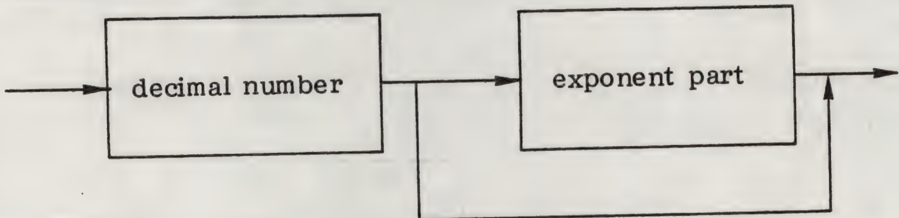
unsigned integer :



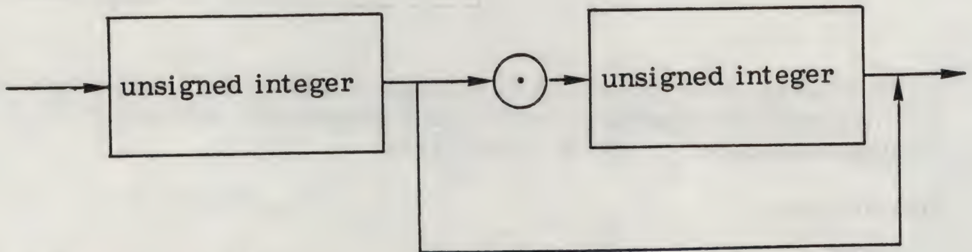
Als in een syntaxdiagram een hulpsymbool voorkomt, is dat hokje te vervangen door het syntaxdiagram van dat hulpsymbool. Aldus kunnen diagrammen samengevoegd worden.

De andere syntaxdiagrammen voor voorgaande syntaxregels (sommige zijn samengevoegd):

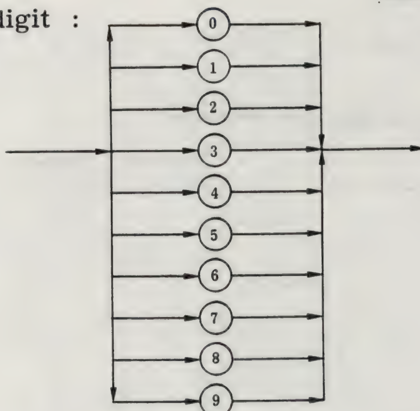
number :



decimal number :



digit :



We sluiten deze paragraaf af met een ander voorbeeld, dat later nog gebruikt zal worden.

Een simpele programmeertaal:

```
<program> ::= <block>$
<block> ::= begin {<declaration>} <statement list> end
<declaration> ::= integer <identifier> {,<identifier>}; |
                  real<identifier>{,<identifier>};
<identifier> ::= <letter>{<digit>|<letter>}
<statement list> ::= <statement>{;<statement>}
<statement> ::= <assignment>|<conditional>|<repetition>|
                <block>|<write>
<assignment> ::= <identifier> := {<identifier> :=}<expression>
<conditional> ::= if <bexpr> then <statement list>
                               [else <statement list>] fi
<repetition> ::= while <bexpr> do <statement list> od
<write> ::= output (<expression>{,<expression>})
<bexpr> ::= <expression><relop><expression>
<relop> ::= < | ≦ | ≧ | > | = | ≠
<expression > ::= <exp1>{<adop><exp1>}
<adop> ::= + | -
<exp1> ::= <exp2>{<mulop><exp2>}
<mulop> ::= * | /
<exp2> ::= input | <identifier>|<number>|(<expression>)
```

We gaan er van uit dat de non-terminals <digit> en <letter> door de voor de hand liggende produktieregels gedefinieerd worden. De non-terminal <number> is reeds gedefinieerd.

3.4 DE CHOMSKY HIËRARCHIE VAN GRAMMATICA'S

De (algemene) grammatica die in 3.2 gedefinieerd is wordt wel een 0-type Chomsky grammatica of onbeperkte Chomsky grammatica genoemd. Door beperkingen op te leggen aan de produktieregels kunnen speciale typen grammatica's gedefinieerd worden.

De grammatica G = (N, T, R, S) is een *context-gevoelige grammatica* (1-type grammatica; Engels: context-sensitive grammar) als iedere produktieregel van R van de vorm $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \omega \alpha_2$ is met $A \in N$, $\alpha_1 \in V^*$, $\alpha_2 \in V^*$ en $\omega \in V^+$ (ten opzichte van een 0-type grammatica geldt nu de beperking dat $\omega \neq \varepsilon$ dus $|\alpha_1 A \alpha_2| \leq |\alpha_1 \omega \alpha_2|$).
L(G) heet een *context-gevoelige taal*.

VOORBEELD 5
G = ({A, B}, {a, b}, {A → aB, aB → aaBB, B → b}, A) is een

context-gevoelige grammatica ($L(G) = \{a^k b^k | k \geqq 1\}$).
$G = (\{A, B, C, S\}, \{a, b,\}, R, S)$ met $R = \{S \to A, A \to aABC, A \to abC, CB \to BC, bB \to bb, bC \to b\}$ is niet context-gevoelig, zie bijvoorbeeld het voorkomen van de produktieregel $bC \to b$. ($L(G) = \{a^k b^k | k \geqq 1\}$). We zien dus dat twee verschillende grammatica's dezelfde taal voortbrengen!
(einde voorbeeld 5)

De grammatica $G = (N, T, R, S)$ heet een *context-vrije grammatica* (2-type grammatica; Engels: context-free grammar) als iedere produktieregel van R van de vorm $A \to \alpha$ is met $A \in N$ en $\alpha \in V^*$.
L(G) heet een *context-vrije taal.*

VOORBEELD 6
$G = (\{S\}, \{a,b\}, \{S \to aSb, S \to ab\}, S)$ is een context-vrije grammatica ($L(G) = \{a^k b^k | k \geqq 1\}$. Dit is de derde grammatica voor die taal!)
(einde voorbeeld 6)

In de definitie van de context-vrije grammatica is toegestaan dat $\alpha = \varepsilon$ (want $\varepsilon \in V^*$). Het maakt geen essentieel verschil of dit wordt toegestaan of niet, want voor iedere context-vrije grammatica G bestaat er een context-vrije grammatica G', waarin ε niet voorkomt, met $L(G') = L(G) \setminus \{\varepsilon\}$.

De grammatica $G = (N, T, R, S)$ heet een *reguliere grammatica* (3-type grammatica; Engels: regular grammar) als òf iedere produktieregel van R van de vorm $A \to aB$ of $A \to a$ is met $A \in N$, $B \in N$ en $a \in T^*$, òf iedere produktieregel van de vorm $A \to Ba$ of $A \to a$ is met $A \in N$, $B \in N$ en $a \in T^*$. Zijn alle produktieregels van de vorm $A \to aB$ of $A \to a$ dan wordt de grammatica ook wel *rechts-lineair* genoemd. (Zo ook voor *links-lineair.*) Een reguliere grammatica is dus of rechts-lineair of links-lineair.

Als beide soorten produktieregels naast elkaar in een grammatica voorkomen, is de grammatica niet regulier.
L(G) heet een *reguliere taal.*

VOORBEELD 7
$G = (\{A, B, S\}, \{0, 1\}, \{S \to 1B, S \to 1, A \to 1B, B \to 0A, A \to 1\}, S)$ is een reguliere grammatica ($L(G) = (10)^*1$). Deze grammatica is rechts-lineair. Dezelfde taal wordt voortgebracht door de links-lineaire grammatica $G_1 = (\{A, B, S\}, \{0, 1\}, \{S \to B1, S \to 1, A \to B1, B \to A0, A \to 1\}, S)$; $L(G_1) = 1(01)^* = (10)^*1 = L(G)$. Dit resultaat geldt algemeen: iedere reguliere taal heeft zowel een links-lineaire als een rechts-lineaire grammatica.
(einde voorbeeld 7)

OPMERKINGEN
Iedere i-type grammatica is ook van het type i-1 (i = 1,2,3). Zo ook is iedere i-type taal ook van het type i-1 (de Chomsky-hiërarchie).
Als een bepaalde taal wordt voortgebracht door een i-type grammatica, kan het zijn dat de taal ook kan worden voortgebracht door een i+1 type grammatica (taal is van het type i+1).
Iedere grammatica definieert één taal; een taal kan echter door meerdere grammatica's gedefinieerd worden.
Er zijn ook grammaticamodellen, die niet binnen de Chomsky-hiërarchie vallen.
Geven we met L_R, L_{CV}, I_{CG}, L_0 en L_M de klassen aan van respectievelijk de reguliere, de context-vrije, de context-gevoelige, de 0-type en de monotone talen aan, dan geldt

$$L_R \subset L_{CV} \subset L_{CG} = L_M \subset L_0$$

We geven hiervan geen bewijs.

3.5 HET HERKENNEN VAN ZINNEN

Tot nu toe is bekeken hoe, via een grammatica, zinnen van een taal gegenereerd kunnen worden. Een belangrijk punt bij programmeertalen is echter dat de zinnen van een taal, de programma's, ook herkend moeten worden opdat ze vertaald kunnen worden naar machinetaal-programma's, waarbij de betekenis gehandhaafd blijft. Dit is het werk van de *vertaler* (compiler).

Het zou plezierig zijn als de regels die gebruikt zijn om de zinnen te genereren ook gebruikt kunnen worden om de zinnen te herkennen. Het zoeken van de toegepaste regels zou echter niet tot een omvangrijk zoekproces moeten leiden.

Stel dat we bij de grammatica uit voorbeeld 4 van § 3.2 willen nagaan of de zin 0112 tot de taal behoort. We kunnen dit in de volgende stappen constateren.
- Om tot 0112 te komen moet de produktieregel S → 0A toegepast zijn. Dit betekent dat 112 vanuit A moet kunnen worden voortgebracht.
- Voor het herschrijven van A zijn twee mogelijkheden aanwezig, doch A → 2 komt niet in aanmerking. Dit betekent dat A → 1A moet zijn toegepast. We moeten dan nog controleren of 12 door A kan worden voortgebracht.
- Dezelfde redenering als hierboven is van toepassing met als conclusie dat 2 door A moet kunnen worden voortgebracht.
- Inderdaad bestaat de produktieregel A → 2.

Hieruit volgt dat de zin 0112 inderdaad voortgebracht wordt door de desbetreffende grammatica.

Laten we nu eens kijken naar de grammatica en de taal van voorbeeld 3 van § 3.2. Stel dat nagegaan moet worden of kkllkk een zin uit de taal is. We kunnen nu niet op dezelfde wijze te werk gaan als bij het vorige voorbeeld. Daar kan de analyse zo eenvoudig plaatsvinden, omdat de grammatica regulier (rechts-lineair) is. Zoals we gezien hebben in § 3.4 is het best mogelijk dat voor een taal, die door een niet-reguliere grammatica is voortgebracht, ook een reguliere grammatica bestaat.

Neem het eerste voorbeeld uit § 3.3. De door die grammatica voortgebrachte taal, de numbers, kan ook gegenereerd worden met behulp van de grammatica:

<number> ::= d<rest number>|.<decimal fraction>|
$_{10}$<exponent part>
<rest number> ::= d<rest number>|.<decimal fraction>|
$_{10}$<exponent part>|ε
<decimal fraction> ::= d<rest decimal fraction>
<rest decimal fraction> ::= d<rest decimal fraction>|
$_{10}$<exponent part>|ε
<exponent part> ::= d<rest unsigned integer>|
+<unsigned integer>
-<unsigned integer>
<unsigned integer> ::= d<rest unsigned integer>
<rest unsigned integer> ::= d<rest unsigned integer>|ε

Deze grammatica is regulier en met deze grammatica kunnen zinnen uit de taal (numbers) op de eerder aangegeven wijze herkend worden. In bovenstaande grammatica staat een regel van de vorm A → dB voor tien regels: A → 0B, A → 1B,..., A → 9B.

We kunnen, voor de herkenning van een zin (number) uit de bovenstaande grammatica een tabel maken waarin vermeld staat hoe vanuit een bepaalde non-terminal (die nog herkend moet worden), via de eerste terminal van de nog te herkennen string, bij de volgende non-terminal gekomen kan worden die daarna nog herkend moet worden. De non-terminals zijn aangegeven met nummers. Een niet-ingevulde plaats in de tabel komt overeen met de situatie, dat de string geen number is volgens de grammatica. Een plaats met 'klaar' geeft aan dat de string herkend is als number. (Zie de tabel op p.39).

Niet iedere grammatica is te herschrijven op de bovengenoemde wijze (alleen grammatica's die reguliere talen genereren). Dus is niet voor alle grammatica's een tabel te maken als de onderstaande. Voor reguliere talen kan wel zo'n tabel gemaakt worden. Zo'n tabel definieert een zogenaamde *eindige automaat*. De non-terminals worden de toestanden van de automaat genoemd en de terminals de invoersymbolen van de automaat.

Vanuit: met terminal:	d	.	10	+ of -	ε
1. <number>	2	3	5		
2. <rest number>	2	3	5		klaar
3. <decimal fraction>	4				
4. <rest decimal fraction>	4		5		klaar
5. <exponent part>	7			6	
6. <unsigned integer>	7				
7. <rest unsigned integer>	7				klaar

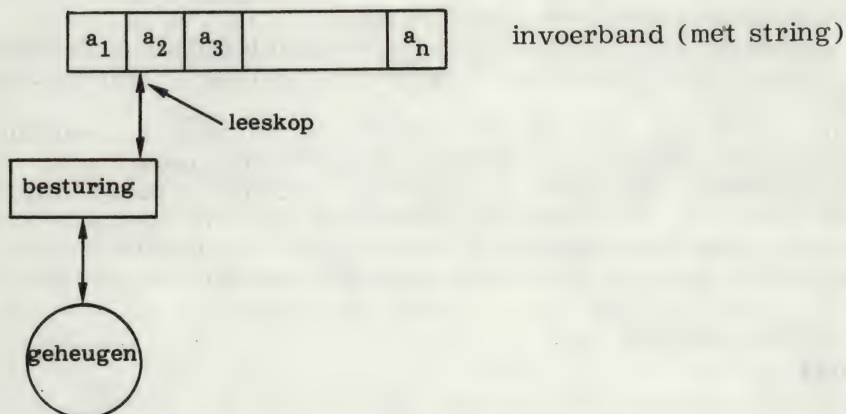
3.6 AUTOMATEN

Het begrip abstracte automaat is geïntroduceerd door A.M. Turing (1936), die het begrip algoritme definieerde met behulp van de naar hem genoemde abstracte automaat: de *Turing machine*. Er zijn drie soorten automaten te onderscheiden: acceptoren, generatoren en omzetters. Een acceptor krijgt een string aangeboden en bepaalt of die string aan een bepaalde eigenschap voldoet of niet. Een generator genereert strings die aan bepaalde eisen voldoen. En een omzetter krijgt een string aangeboden en produceert een andere string. We zullen ons hier beperken tot acceptoren.

Zo'n acceptor moet gemaakt worden als we een compiler bouwen. Belangrijk hierbij zijn de vragen:
- voor welke talen is dit mogelijk, en
- wat voor automaat hoort bij een bepaald type taal?

De theorie der abstracte automaten geeft hierop antwoord.

Een *acceptor* is een abstract mechanisme, dat we als volgt in beeld kunnen brengen:



De *invoerband* bestaat uit vakjes en bevat een string, waarbij ieder symbool van de string in een apart vakje staat. Eventueel kan de invoerband rechts en links afgesloten worden door een speciaal eindsymbool. De band kan naar links bewogen worden langs de *leeskop*.

Het *geheugen* kan symbolen uit een bepaald alfabet bevatten (het geheugen is onbeperkt groot). Het gedrag van het geheugen wordt bepaald door de wijze waarop deze symbolen opgeslagen en geaccesseerd kunnen worden. Zo kan het geheugen een stapel-gedrag vertonen. Juist het geheugengedrag bepaalt het type van de acceptor.

De *besturing* kan opgevat worden als een programma dat bepaalt hoe de automaat reageert. Dit programma kan gezien worden als een eindige verzameling toestanden en een afbeelding die uit de heersende toestand, het huidige invoersymbool en de informatie uit het geheugen een nieuwe toestand bepaalt (en of er gelezen moet worden en welke gegevens in het geheugen geplaatst moeten worden).

De *configuratie* van de automaat (de totale toestand) wordt op ieder moment bepaald door: de toestand van de besturing, de invoerstring en de plaats van de leeskop en de inhoud van het geheugen.

De automaat wordt *non-deterministisch* genoemd als vanuit een toestand meerdere opvolgende toestanden mogelijk zijn; de automaat is *deterministisch* als steeds slechts één opvolgende toestand mogelijk is.

De *begintoestand* van de automaat wordt gekenmerkt door een speciale toestand van de besturing, de leeskop staande op het meest linkse symbool van de invoerstring en eventueel een speciale inhoud van het geheugen. In een *eindtoestand* geldt:

- de besturing bevindt zich in één van een aantal speciale toestanden;
- de leeskop staat op het rechter eindsymbool (de band met de string is geheel gelezen);
- het geheugen heeft een bepaalde inhoud.

We zeggen dat de acceptor de string ω accepteert als met ω op de invoerband de acceptor vanuit de beginstoestand in een eindtoestand terecht komt.

De *taal die de acceptor definieert* is de verzameling van alle invoerstrings die door de acceptor geaccepteerd worden.

Zoals er een hiërarchie is voor de grammatica's (en de bijbehorende talen), zo is er ook een hiërarchie voor de abstracte automaten en deze twee komen met elkaar overeen. De eindige automaat is een acceptor die overeenkomt met reguliere grammatica's.

De eindige automaat

DEFINITIE

Een *eindige automaat* M over een alfabet T is een geordend vijftal

(K, T, δ, q_0, F), met
- K is een eindige, niet lege verzameling van toestanden (van de besturing),
- T is het alfabet van invoersymbolen,
- δ is een afbeelding van K*T in K,
- q_0 is de begintoestand, $q_0 \in K$,
- F is de verzameling van eindtoestanden, $F \subset K$.

(einde definitie)

Er is in dit geval geen geheugen.
We zullen niet ingaan op een eventuele fysieke representatie van deze abstracte automaat.

De automaat verkeert initieel in toestand q_0. Het meest 'linkse' symbool van de invoerstring (die een element is van T*) wordt 'gelezen'; de automaat gaat over in de toestand $\delta(q_0,a)$ als a het gelezen invoersymbool is, en het volgende symbool uit de invoerstring kan gelezen worden. Als dit symbool b is, is de volgende toestand $\delta(\delta(q_0,a),b)$.

We kunnen de afbeelding δ, die nu gedefinieerd is als $K*T \rightarrow K$ uitbreiden tot δ': $K*T^* \rightarrow K$ door middel van

$\delta'(q,\varepsilon) = q$

$\delta'(q,xa) = \delta(\delta'(q,x),a)$ voor $x \in T^*$ en $a \in T$.

Vaak worden δ en δ' door elkaar gebruikt.

Een string x wordt geaccepteerd door M als $\delta(q_0,x) = p$ met $p \in F$.

DEFINITIE

De verzameling van alle strings die door M geaccepteerd worden, wordt aangegeven met A(M): $A(M) = \{x \mid x \in T^*, \delta(q_0,x) \in F\}$.

(einde definitie)

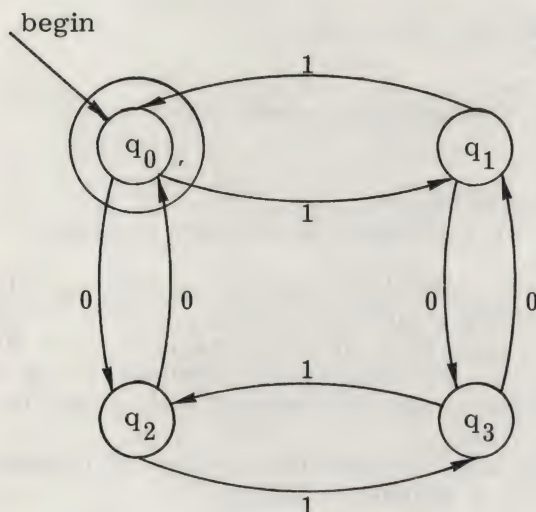
VOORBEELD 8

$M = (\{q_0, q_1, q_2, q_3\}, \{0,1\}, \delta, q_0, \{q_0\})$ met $\delta(q_0,0) = q_2$, $\delta(q_0, 1) = q_1$, $\delta(q_1, 0) = q_3$, $\delta(q_1, 1) = q_0$, $\delta(q_2, 0) = q_0$, $\delta(q_2, 1) = q_3$, $\delta(q_3, 0) = q_1$ en $\delta(q_3, 1) = q_2$.

We kunnen de afbeelding δ met behulp van een tabel, de zogenaamde toestandstabel (zie ook § 3.5), aangeven:

δ	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Vaak ook wordt een zogenaamd *toestandsdiagram* gegeven:

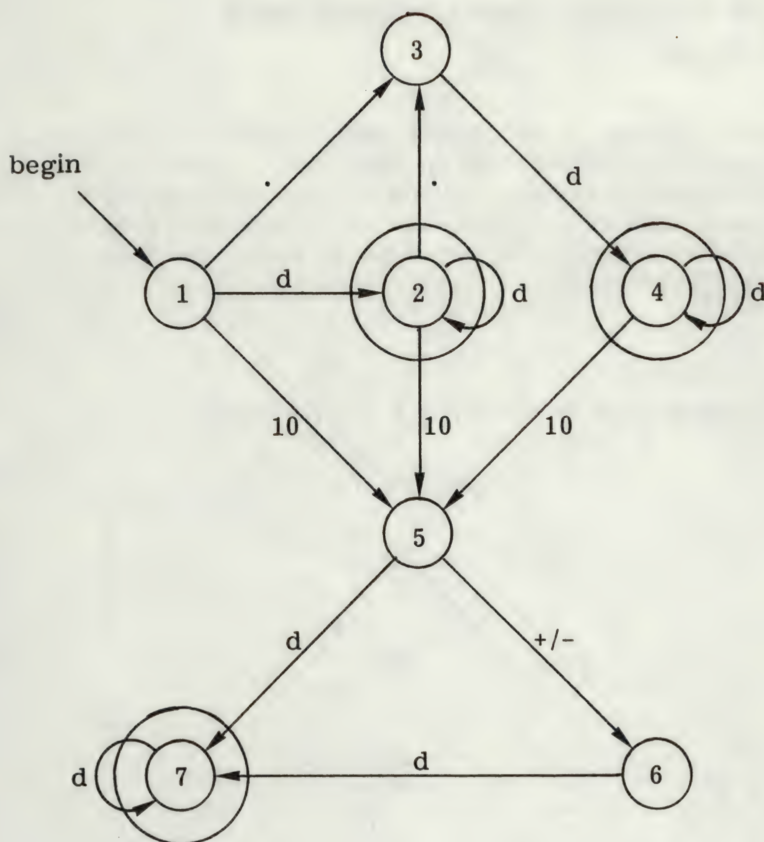


De begintoestand wordt aangegeven door een verwijzing. De eindtoestanden worden omcirkeld. Een pijl met een invoersymbool geeft aan dat de ene toestand onder invloed van het invoersymbool overgaat in de andere toestand.

We zien dat voor deze M geldt dat bijvoorbeeld 0101, 11 en 011000 geaccepteerd worden. In feite worden alle elementen van $\{0, 1\}^*$ geaccepteerd, die bestaan uit een even aantal enen en een even aantal nullen.
(einde voorbeeld 8)

Voor de automaat die de numbers, die gegenereerd worden door de grammatica uit § 3.5, herkent is de automaat eigenlijk al gegeven in tabelvorm.

Het toestandsdiagram is gegeven op p.43. De ε-regels zijn niet opgenomen, maar als het ware verwerkt in de eindtoestanden.



Zonder bewijs geven we nu de belangrijke *STELLING*:

Zij G = (N, T, R, S) een reguliere grammatica. Er bestaat een eindige automaat M = (K, T, δ, q_0, F) zodanig dat A(M) = L(G). Zo bestaat er ook bij een gegeven eindige automaat M een reguliere grammatica G zodanig dat L(G) = A(M).
(einde stelling)

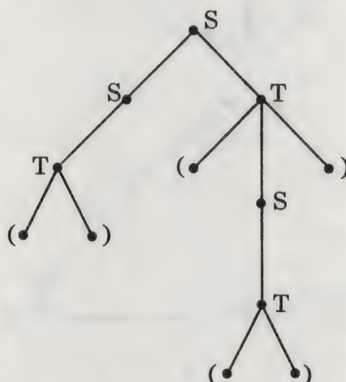
Je kunt niet alleen de existentie van M (of G) aantonen, je kunt deze zelfs construeren (afleiden). Bij iedere reguliere grammatica is dus een eindige automaat te construeren die de zinnen uit de taal, gegenereerd door die grammatica, kan herkennen. Van een willekeurige string kan nagegaan worden of het een zin is uit die taal. Dit is van belang bij het maken van een vertaler.

3.7 SYNTACTISCHE BOMEN VOOR CONTEXT-VRIJE GRAMMATICA'S

In een grammatica kunnen verschillende afleidingen *equivalent* zijn, dat wil zeggen dat in de afleidingen op dezelfde plaatsen dezelfde produktieregels worden toegepast, alleen in verschillende volgordes. Equivalente afleidingen worden door dezelfde *syntactische boom* (derivatieboom) voorgesteld (we tekenen bomen met de wortel boven en de bladeren onder).

VOORBEELD 9
G = ({S,T}, {(,)},{S →ST,S →T, T → (S), T → ( )},S)

De syntactische boom voor de afleiding $S \overset{*}{\Rightarrow}$ ( ) (( )) is:



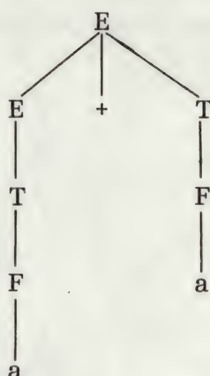
(einde voorbeeld 9)

Bij iedere afleiding in een context-vrije grammatica hoort een syntactische boom. Verschillende afleidingen kunnen dezelfde boom geven (de afleidingen zijn equivalent). Als nu nagegaan moet worden of een zin tot de taal behoort, en dat gebeurt door een afleiding te zoeken, kunnen veel afleidingen geprobeerd worden. Om al dit proberen wat gerichter te doen kan geprobeerd worden op een systematische manier de syntactische boom te construeren. Dit kan bijvoorbeeld door in de afleiding steeds een produktieregel toe te passen op de non-terminal die het meest links (of rechts) in de zinsvorm voorkomt.

Een afleiding $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$ in een context-vrije grammatica heet een *links-afleiding* (Engels: leftmost derivation) als voor iedere i ($0 \leq i < n$) geldt $\alpha_i = xA\omega \Rightarrow x\beta\omega = \alpha_{i+1}$ met $(A \rightarrow \beta) \in R$, $x \in T^*$ en $\omega \in V^*$.

Op analoge wijze kan een *rechts-afleiding* gedefinieerd worden.

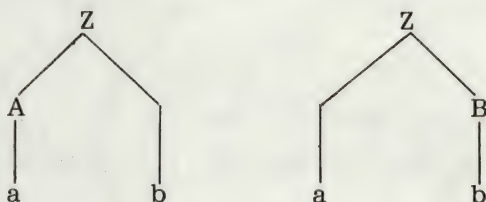
VOORBEELD 10
Laat G zijn: ({E,F,T}, {+,(,*,)}, {E → E+T, E → T, T → T * F, T → F, F → (E), F → a}, E).
Er zijn tien afleidingen voor de zin a + a uit E, die allemaal gerepresenteerd worden door de syntactische boom



De links-afleiding is : E ⇒ E+T ⇒ T+T ⇒ F+T ⇒ a+T ⇒ a+F ⇒ a+a.
De rechts-afleiding is: E ⇒ E+T ⇒ E+F ⇒ E+a ⇒ T+a ⇒ F+a ⇒ a+a.
Een afleiding die noch links noch rechts is:
E ⇒ E+T ⇒ T+T ⇒ T+F ⇒ F+F ⇒ a+F ⇒ a+a.
(einde voorbeeld 10)

Een zin die verschillende syntactische bomen heeft wordt *dubbelzinnig* (*ambigu*; Engels: ambiguous) genoemd. We kunnen ook zeggen dat de zin verschillende links-afleidingen heeft (steeds binnen dezelfde grammatica).

VOORBEELD 11
G = ({A, B, Z}, {a, b}, {Z → Ab, Z → aB, A → a, B → b, A → Aa, B → Bb}, Z). De zin ab is dubbelzinnig:

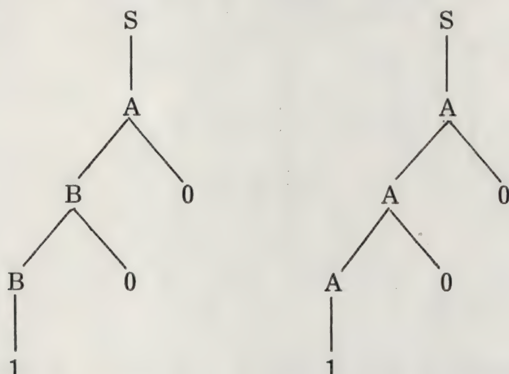


Links-afleidingen: Z ⇒ Ab ⇒ ab en Z ⇒ aB ⇒ ab.
(einde voorbeeld 11)

Een context-vrije grammatica is *dubbelzinnig* als door de grammatica ten minste één dubbelzinnige zin wordt voortgebracht.

Ook al hebben de bomen dezelfde vorm, toch kan dubbelzinnigheid optreden.

VOORBEELD 12
G = ({A, B, S}, {0, 1}, {S → A, A → B0, A → A0, B → B0, A → 1, B → 1}, S).
De zin 100 is in deze grammatica dubbelzinnig.



De vormen van de bomen zijn weliswaar gelijk, doch de bomen zelf (en ook de links-afleidingen) zijn verschillend.
(einde voorbeeld 12)

Een taal is *dubbelzinnig* als de taal alleen voortgebracht kan worden door dubbelzinnige grammatica's.

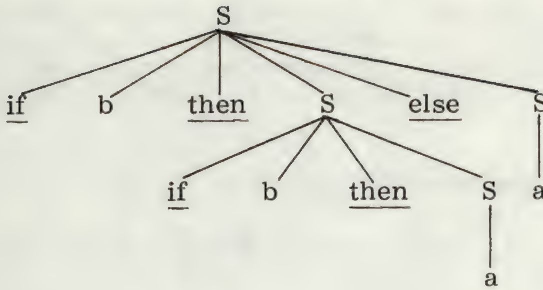
Ook bij het definiëren van programmeertalen wil men uiteraard dubbelzinnigheid vermijden. Een bekend voorbeeld van dubbelzinnigheid, die door extra maatregelen kan worden voorkomen, is een gevolg van de produktieregels:

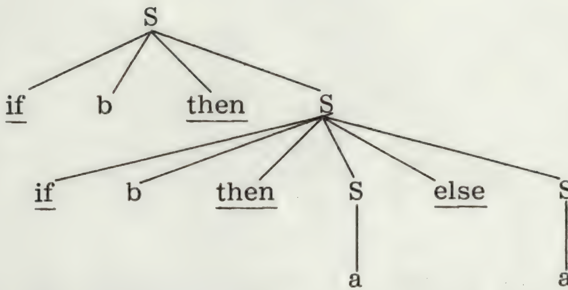
S → if b then S else S
S → if b then S
S → a

De zin

if b then if b then a else a

is dubbelzinnig, omdat er twee syntactische bomen (twee links-afleidingen) voor zijn:





Bij het vertalen wordt eerst de syntactische boom gecreëerd om van daaruit tot de generatie van code te komen.

3.8 STAPELAUTOMATEN

We hebben reeds gezien dat een acceptor gebruikt kan worden om na te gaan of een bepaalde string tot de beschouwde taal behoort of niet. In § 3.6 is algemeen over acceptoren gesproken en is de eindige automaat als voorbeeld getoond.

De acceptor die overeenkomt met een context-vrije grammatica blijkt de eigenschap te hebben dat het geheugen een *stapel* (Engels: stack) is en wordt daarom een stapelautomaat (Engels: pushdown acceptor) genoemd.

We beginnen met een voorbeeld.

We gaan uit van de context-vrije grammatica

$$G = (\{S\}, \{a,b,c\}, \{S \rightarrow aSa, S \rightarrow bSb, S \rightarrow c\}, S).$$

We willen nagaan of de string ababbcbbaba tot de taal L(G) behoort. Daartoe wordt de string van links naar rechts gelezen. Het eerste symbool is een a; de enige mogelijkheid waardoor deze a gegenereerd kan zijn, is via de produktieregel S → aSa. Wil de string tot

L(G) behoren, dan moet dus babbcbbaba van de vorm Sa zijn. Het tweede symbool wordt gelezen (een b) en de string behoort tot L(G) als abbcbbaba van de vorm Sba is. Dit proces wordt voortgezet tot de hele invoerstring verwerkt is; op dat moment moet er niets meer te herkennen zijn.

We kunnen het herkenningsproces weergeven als:

nog te herkennen	niet verwerkt deel van invoerstring
S	ababbcbbaba
Sa	babbcbbaba
Sba	abbcbbaba
Saba	bbcbbaba
Sbaba	bcbbaba
Sbbaba	cbbaba
bbaba	bbaba
baba	baba
aba	aba
ba	ba
a	a

De rij van nog te herkennen symbolen gedraagt zich inderdaad als een stapel, waarbij in dit geval het 'bovenste' element steeds wordt vervangen door twee elementen.

In dit voorbeeld is aan de hand van de combinatie van de top van de stapel en het invoersymbool altijd direct te bepalen wat er dient te gebeuren (door de grammatica; zouden naast elkaar voorkomen de produktieregels S → aSa en S → aSb, dan was dit niet het geval). De automaat bij de gegeven grammatica is *deterministisch*.

Het kan voorkomen dat een bepaalde operatie moet worden uitgevoerd zonder dat er een invoersymbool wordt verwerkt. Neem de grammatica voor aritmetische expressies met als produktieregels: E → E+T, E → T, T → T * F, T → F, F → (E), F → a. Bij de herkenning van een string als (a+a)*a zal E op de stapel vervangen moeten worden door E+T zonder dat er een invoersymbool wordt verwerkt.

Let op het verschil met de eindige (deterministische) automaat. Daar hoefde bij de stappen in de herkenning niets onthouden te worden. Hier wordt in de stapel onthouden: de nog niet herkende delen van rechterleden van reeds gebruikte produktieregels.

DEFINITIE

Een *stapelautomaat* M is een geordend zestal (K,T,V,P,I,F), waarvoor geldt:

- K is een eindige verzameling waarvan de elementen toestanden worden genoemd;
- T is het invoeralfabet ($T \subset V$);
- V is het stapelalfabet;

- P is het 'programma' van M;
- I is de verzameling van begintoestanden ($I \subset K$);
- F is de verzameling van eindtoestanden ($F \subset K$).

(einde definitie)

De opdrachten in P zijn van één van de volgende vormen:
- lees invoersymbool en ga over in nieuwe toestand;
- zet symbool op de stapel en ga over in nieuwe toestand;
- haal topelement van stapel en ga over in nieuwe toestand.

Als q een toestand is ($q \in K$), x een string is ($x \in T^*$) en α een stapelstring is ($\alpha \in V^*$), dan wordt (q,x,α) een *configuratie* van M genoemd. Een configuratie bepaalt volledig de totale toestand van M tijdens het herkennen van een invoerstring. De configuratie (q,x,α) betekent dat de toestand q heerst, dat van de totale invoerstring y het gedeelte x is 'gelezen' (de leeskop staat op het laatste symbool van x) en de inhoud van het geheugen is α. We zullen het effect van de bovenstaande drie typen opdrachten uitdrukken in veranderingen van de configuratie.

- Als de configuratie (q,x,α) is en de invoerstring y is gelijk aan xaz, dan heeft de leesopdracht tot gevolg dat het volgende symbool (a) gelezen wordt en dat M overgaat in toestand q'. Dus:

  $(q,x,\alpha) \overset{L}{\rightarrow} (q',xa,\alpha)$.

  We geven deze opdracht aan met L(q,a,q').
- Het zetten van symbool U ($\in V$) op de stapel in de configuratie (q.x.α) heeft de nieuwe configuratie $(q',x,\alpha U)$ tot gevolg:

  $(q,x,\alpha) \overset{S}{\rightarrow} (q',x,\alpha U)$.

  We geven deze opdracht weer als S(q,U,q').
- Als de configuratie (q,x,α) is met $\alpha = \beta U$, dan leidt het weghalen van het topelement van de stapel tot de configuratie (q',x,β):

  $(q,x,\beta U) \overset{U}{\rightarrow} (q',x,\beta)$.

  We geven deze opdracht aan met U(q,U,q').

Bij iedere configuratie is slechts één van deze soorten opdrachten mogelijk. Wel zijn (als de automaat non-deterministisch is) verschillende opdrachten van dezelfde soort mogelijk bij een bepaalde toestand.

Als M door middel van uitvoering van opdrachten de volgende rij configuraties doorloopt $(q_0, x_0, \alpha_0) \rightarrow (q_1, x_1, \alpha_1) \rightarrow \ldots \rightarrow (q_k, x_k, \alpha_k)$ dan schrijven we $(q_0, x_0, \alpha_0) \Rightarrow (q_k, x_k, \alpha_k)$.

Bij aanvang bevindt de automaat zich in een van de begintoestanden, het geheugen is leeg en de leeskop bevindt zich op het linker afsluitsymbool van de invoerband. De automaat doorloopt een aantal configuraties, door uitvoering van opdrachten, waarbij iedere volgende configuratie wordt bepaald door de opdracht die in de vorige

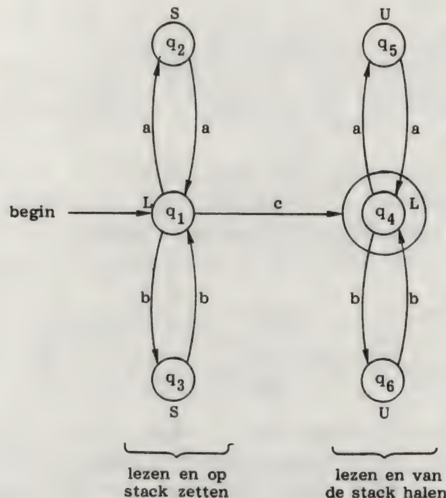
configuratie mogelijk is. De automaat stopt als een configuratie bereikt is waarin geen opdracht meer mogelijk is. Als alle symbolen van de invoerstring x zijn gelezen, de stapel leeg is en de besturing zich in één van de eindtoestanden bevindt, zeggen we dat de string x door M *geaccepteerd* is.

De *beginconfiguratie* van M is van de vorm $(q, \varepsilon, \varepsilon)$ met $q \in I$. De *eindconfiguratie* van M is van de vorm (q, x, ε) met $q \in F$ en x is 'voorste' deel van de invoerstring y. Als x = y is de string y geaccepteerd en dan geldt $(q_i, \varepsilon, \varepsilon) \Rightarrow (q_e, y, \varepsilon)$ met $q_i \in I$ en $q_e \in F$.

DEFINITIE
De *taal* die door M wordt gedefinieerd (de door M geaccepteerde strings) is $A(M) = \{x \in T^* | (q_i, \varepsilon, \varepsilon) \Rightarrow (q_e, x, \varepsilon)$ met $q_i \in I, q_e \in F\}$
(einde definitie)

De automaat van ons voorbeeld heeft de vorm $M = (\{q_1,q_2,q_3,q_4,q_5,q_6\}, \{a,b,c\},\{a,b\}, P, \{q_1\},\{q_4\})$ met voor P: $L(q_1,a,q_2)$, $L(q_1,b,q_3)$, $L(q_1,c,q_4)$, $S(q_2,a,q_1)$, $S(q_3,b,q_1)$, $L(q_4,a,q_5)$, $L(q_4,b,q_6)$, $U(q_5,a,q_4)$ en $U(q_6,b,q_4)$. In toestand q_1 wordt een invoersymbool gelezen; is dit een a dan gaat M over in toestand q_2, bij een b volgt q_3 en bij een c volgt q_4. Vanuit $q_2(q_3)$ wordt het eerste gedeelte van de invoerstring gelezen en op de stapel gezet. Als de c wordt gelezen gaat de automaat over in q_4 en de rest van de invoerstring wordt gelezen en vergeleken met de stapel (de toestanden q_4, q_5 en q_6).

Het toestandsdiagram ziet er uit als:



Bij de toestanden is aangegeven welk type opdracht mogelijk is.

We zien dat in elke configuratie precies één opdracht kan worden uitgevoerd, de automaat is deterministisch. M accepteert iedere string van de vorm xcx^R met $x \in \{a,b\}^*$ (x^R is de string die uit x wordt verkregen door de omgekeerde volgorde te nemen). Het accepteren van abcba verloopt als volgt:

$(q_1,\varepsilon,\ \varepsilon) \overset{L}{\rightarrow} (q_2,a,\varepsilon) \overset{S}{\rightarrow} (q_1,a,a) \overset{L}{\rightarrow} (q_3,ab,a) \overset{S}{\rightarrow} (q_1,ab,ab) \overset{L}{\rightarrow}$
$(q_4,abc,ab) \overset{L}{\rightarrow} (q_6,abcb,ab) \overset{U}{\rightarrow} (q_4,abcb,a) \overset{L}{\rightarrow} (q_5,abcba,a) \overset{U}{\rightarrow}$
$(q_4,abcba,\varepsilon)$.

Door de definitie van de automaat hebben we op de stapel de S, die we bij het begin van deze paragraaf in het voorbeeld gebruikten, niet meer nodig.

Een zeer belangrijke *STELLING* luidt:

Als L een context-vrije taal is met grammatica G, dan is er een stapelautomaat M te construeren met A(M) = L(G). Zo ook: Als M een stapelautomaat is, dan is A(M) een context-vrije taal.
(einde stelling)

Het belang van deze stelling is weer dat er bij een gegeven context-vrije grammatica een automaat geconstrueerd kan worden die kan nagaan of een zin door deze grammatica is gegenereerd. Als de automaat door een algoritme is te representeren, is daardoor een deel van de vertaler gerealiseerd.

4 SEMANTIEK

4.1 INLEIDING

In het vorige hoofdstuk is aandacht besteed aan de syntaxis. Wil men programmeertalen beoordelen of vergelijken, dan kan uiteraard de syntaxis van de talen bestudeerd worden. Hieruit valt dan te leren welke constructiemogelijkheden in de taal aanwezig zijn. Dit geeft al enige indicatie over de bruikbaarheid van de taal. Met die constructies valt echter weinig te beginnen als de semantiek ervan niet bekend is. Wil een constructie verantwoord gebruikt kunnen worden, dan moet het effect van die constructie bekend zijn. Daarnaast komt nog het pragmatische aspect aan de orde: hoe moeilijk of hoe gemakkelijk is het datgene, dat uitgedrukt moet worden, in het programma vast te leggen? Als programmeertalen vergeleken worden moet dan ook niet alleen aandacht besteed worden aan de syntaxis maar zeker ook aan de semantiek en de pragmatiek. In het programmeeronderricht is de nadruk, die op de syntaxis wordt gelegd, duidelijk waar te nemen. Men kiest voor (een subset van) een bestaande programmeertaal op grond van beschikbaarheid of notatie. Men gaat niet vaak (genoeg) uit van de relatie tussen pragmatiek en semantiek om op grond daarvan voor een bepaalde taal te kiezen.

Voor de syntaxis is in hoofdstuk 3 een formele definitiewijze geïntroduceerd. Moet en kan de semantiek ook formeel behandeld worden?

Van de taal ALGOL 60 is de semantiek zeer zorgvuldig gedefinieerd in een natuurlijke taal (Engels). Ook al is dit zorgvuldig gebeurd, toch is deze definitie op sommige plaatsen incompleet, dubbelzinnig en moeilijk te begrijpen. Daarnaast spelen nog de moeilijkheden van de implementeerbaarheid. Al deze mogelijke problemen worden natuurlijk niet automatisch vermeden als we de semantiek formeel definiëren, wel is de kans hierop groter. Een moeilijkheid is wel, zoals bij elk formeel systeem, dat we ons vertrouwd moeten maken met de gebruikte notatie.

De studie van de semantiek stelt ons in staat:

- programmeertalen gemakkelijker te leren;

- een beter begrip te krijgen voor de verschillende constructies en beter verschillende notaties te kunnen vergelijken;
- betere talen te definiëren.

Er zijn drie redenen waarom de semantiek (formeel) zou moeten worden vastgelegd.

1. De implementator van de taal op de machine moet de semantiek kunnen vastleggen in de compiler en kunnen aantonen dat dit correct is gebeurd.
2. Ook voor de ontwerper van een taal is het belangrijk, omdat bij het ontwerp moet worden uitgegaan van de semantiek en de pragmatiek. Als de semantiek formeel wordt vastgelegd kan de ontwerper wellicht ook begrippen beter omschrijven. In het verleden speelde bij een taalontwerp de syntaxis vaak een te grote en allesbeheersende rol.
3. Zoals we al gezien hebben moet de programmeur precies weten wat het effect is van de constructies die hij gebruikt. Bovendien moet hij/zij de correctheid van het programma kunnen aantonen.

Zoals al eerder gezegd is de scheidslijn tussen syntaxis en semantiek niet evident. Dat bijvoorbeeld een variabele gedeclareerd moet worden voor het gebruik kan men opvatten als een semantische kwestie, men kan het ook zien als een regel in een context-gevoelige grammatica.

Men probeert ook wel een deel van de betekenis in de syntaxis vast te leggen. Zo zal bijvoorbeeld in de syntaxis van aritmetische expressies, door het gebruik van de juiste hulptermen op de goede plaats in de produktieregels, al opgesloten zitten dat vermenigvuldigen een hogere prioriteit heeft dan optellen.

Er zijn ook grammatica's die juist bedoeld zijn om ook een stuk betekenis vast te leggen. Neem de volgende regels:

<unsigned integer> ::= <digit>
waarde(<unsigned integer>) =
waarde(<digit>)
| <unsigned integer>$_2$ <digit>
waarde(<unsigned integer>) =
10*waarde(<unsigned integer>$_2$)
+ waarde(<digit>)

<digit> ::= 0
waarde(<digit>) = 0
| 1
.....
| 9
waarde(<digit>) = 9

Ieder alternatief in een produktieregel gaat gepaard met een regel die de betekenis (de waarde) vastlegt. Men noemt een grammatica

met dit soort regels een *attribuut-grammatica*; de betekenis wordt vastgelegd met behulp van de zogenaamde attributen.

Zo zou men ook in de syntaxis niet alleen de programma's kunnen definiëren, maar ook de drietallen (invoer, programma, uitvoer). Ook dan legt men in de syntaxis de betekenis vast: een bepaalde invoer en programma leggen de uitvoer dan vast.

Nu terug naar de formele semantiek.

Een onderwerp waarbij een formele semantiekbeschrijving uiterst belangrijk is, is het probleem van de equivalentie van programma's. (Bijvoorbeeld: een programma in een hogere taal en de vertaling daarvan in machinetaal.)

De relatie tussen semantiek en implementatie is een controversieel onderwerp. Er kan uitgegaan worden van de semantiek om daarop de implementatie te baseren. Men kan ook voor constructies evaluatiemechanismen specificeren, zodat het zeker is dat de constructie geïmplementeerd kan worden, en daarop de semantiek baseren. Deze redenering voortzettend kom je tot de uitspraak dat de vertaling van een taal naar een onderliggende taal (of machine) de betekenis vastlegt. Een bezwaar hiervan is dat op deze manier niet alleen de taal maar ook zijn implementatie gedefinieerd wordt. Van de andere kant heeft het vaak geen zin om over constructies en hun betekenis te spreken als deze constructies niet geïmplementeerd kunnen worden.

De genoemde redenen voor de (formele) beschrijving van de semantiek vertegenwoordigen ieder een verschillende kijk op de programmeertaal (implementator, taalontwerper, programmeur). Deze benaderingen kunnen zo verschillend zijn, dat niet is te volstaan met één enkele semantiekdefinitie (bijvoorbeeld: Hoare's semantiekbeschrijving is niet bij uitstek een hulpmiddel voor de taalontwerper of de implementator). Worden er echter meerdere semantiekdefinities gegeven, dan moet wel aangetoond worden dat deze definities equivalent zijn.

Op het gebied van de semantiek is er de laatste jaren veel te doen geweest. In de literatuur zijn vele voorstellen voor de semantiekbeschrijving gegeven. Naast de babylonische ontwikkeling op het gebied van programmeertalen vindt eenzelfde ontwikkeling plaats met betrekking tot metatalen. De verschillende aanpakken kunnen grofweg in twee categorieën verdeeld worden:

- De vertaler-georiënteerde aanpak.
  Bij deze aanpak wordt de betekenis uitgedrukt in termen van vertalingen naar een abstracte representatie van de syntactische structuur (bijvoorbeeld een syntactische boom) van programma's in de taal.
- De interpretator-georiënteerde aanpak.
  Bij deze aanpak wordt de betekenis vastgelegd in termen van de transformaties die vastgelegd zijn in programma's die volgens de

desbetreffende programmeertaal syntactisch correct zijn. De programmeerteksten verschijnen min of meer direct in de definities. We zullen ons beperken tot modellen die tot deze categorie behoren. Bij deze categorie geven we aan wat de metataal is waarin we de semantiek vastleggen (machine-toestanden, functies, lambda-expressies, proposities) en hoe de constructies uitgedrukt kunnen worden in transformaties van de objecten van de beschrijvingstaal.

We kunnen de methoden die behoren tot de interpretator-georiënteerde semantiekdefinities weer classificeren:
- de operationele methoden,
- de denotationele methoden,
- de axiomatische methoden.

We zullen de drie soorten methoden in het kort bespreken en aan de hand van de mini-taal uit hoofdstuk 3 (laatste deel van § 3.3) enkele voorbeelden laten zien.

4.2 DE OPERATIONELE SEMANTIEK

Een operationeel model van een programmeertaal wordt gegeven door:
- de definitie van abstracte machine-toestanden S, waarin de essentiële informatie over de voortgang van een proces, behorende bij een programma in die taal, wordt vastgelegd;
- de beschrijving van de betekenis van de constructies uit die taal in termen van hun effect op de machinetoestand, dus als een toestandstransformerende functie: $S \rightarrow S$.

Deze beschrijvingswijze is voor veel programmeertalen toegepast. Een bekend voorbeeld hiervan is de semantiekdefinitie voor PL/I. In dit geval worden de toestanden beschreven door boomvormige structuren.

In ons voorbeeld bestaat de toestand uit de zogenaamde toestandsvector, die van iedere variabele uit het programma de heersende waarde bevat. De functie voor een toestandsovergang kan resulteren in een rij van toestanden, die de tussentoestanden van het proces voorstellen. Deze tussentoestanden worden beschreven, ook al is men alleen geïnteresseerd in de laatste toestand van de rij, die het uiteindelijke effect representeert. De semantiekdefinitie wordt gegeven in termen van de functie 'betekenis' met twee argumenten, het eerste is de programmatekst en het tweede is de heersende toestand. Daarnaast wordt gebruik gemaakt van twee hulpfuncties:
- uit; deze levert de laatste toestand af van een eindige rij van toestanden en is ongedefinieerd als de rij niet eindig is;

- bereken; deze geeft de waarde van een expressie bij een gegeven toestand.

Uit de toestand s = (s_1, s_2, ..., s_n) wordt een waarde geselecteerd door s(x), waarbij x een naam van een variabele is. Als in de toestand s een nieuwe waarde voor een variabele moet worden opgenomen, noteren we dit als s[x ← v] met als betekenis s[x ← v](x) = v en s[x ← v](y) = s(y) voor x ≠ y.

VOORBEELDEN

- assignment:
  betekenis(x := e, s) = s[x ← bereken(e, s)]
- statement list:
  betekenis(st_1; st_2, s) = betekenis(st_1, s) ||
  betekenis(st_2, uit(betekenis(st_1, s))
  waarbij || wordt gebruikt voor de concatenatie van rijen toestanden
- repetition:

```
betekenis(while b do stl od, s) =
        if bereken(b,s)
           then betekenis(stl, s) ||
                    betekenis(while b do stl od,
                              uit(betekenis(stl, s)))
           else s
        fi
```

(einde voorbeelden)

Met behulp van de operationele definitie kunnen we uitspraken doen over de correctheid van een programma. Neem het volgende stukje programma, dat bij gegeven x (≧ 0) de waarde van x! berekent en deze waarde toekent aan z:

```
y := 0;
z := 1;
while y ≠ x do y := y + 1;
               z := z * y
            od
```

Om de correctheid van het programma aan te tonen leggen we het gewenste effect vast in een uitspraak over toestanden en bewijzen dat deze gelijk is aan: uit(bereken(programma, s)). De uitspraak hier is:

s(z) = s(x)!

De rij van toestanden is s_0, s_1, ..., s_k, waarbij

s_0 = uit(betekenis(y := 0; z := 1, s))
s_{i+1} = uit(betekenis(y := y+1; z := z*y, s_i))

Om aan te tonen dat in de eindtoestand s_k geldt dat s_k(z) = s_k(x)!,

tonen we aan dat voor iedere toestand s_i geldt dat $s_k(z) = s_i(y)!$. Omdat uit de definitie van de repetitie bekend is dat eindtoestand s_k alleen bereikt kan worden als $s_k(y) = s_k(x)$, weten we dan dat $s_k(z) = s_k(x)!$.
(Voor het bewijs van de correctheid wordt gebruik gemaakt van volledige inductie.)

```
    s0 = uit(betekenis(y := 0; z := 1, s))
       = s[y ← 0] || betekenis(z := 1; uit(s[ y ← 0]))
       = s[y ← 0] [z ← 1]
```

maar 0! = 1 dus $s_0(z) = s_0(y)!$
Stel $s_i(z) = s_i(y)!$

```
    si+1 = uit(betekenis(y := y+1; z := z*y, si))
         = uit(betekenis(z := z*y, si[y ←  bereken(y+1, si)]))
         = si[y ← bereken(y+1, si)]
                  [z ← bereken(z*y, si[y ← bereken(y+1, si)])]
         = si[y ← bereken(y+1, si)]
                  [z ← bereken((y+1)*y!, si)]
```

En dus $s_{i+1}(z) = s_{i+1}(y)!$.
Hieruit volgt dan $s_k(z) = s_k(y)!$ (en $s_k(y) = s_k(x)$).

Dat het programma werkelijk eindigt vraagt een apart bewijs.

4.3 DE DENOTATIONELE SEMANTIEK

Het woord denotatie bestaat eigenlijk niet in het Nederlands. Het Engelse woord denotation betekent zoiets als 'de naam voor', 'aanduiding'. Wij zullen toch het woord denotatie gebruiken en zeggen dat bijvoorbeeld 4 + 2 en 2 * 3 denotaties zijn voor de waarde 6. Bij de vorige definitiemethode werd gesproken over toestanden en toestandsovergangen. Bij de denotationele methode wordt de betekenis op een abstracter niveau gedefinieerd. Het programma wordt niet als een transformator van toestanden gezien maar als een functie. De betekenis wordt dus niet uitgedrukt in rijen toestanden, maar in functies van toestanden in toestanden.

De vroegste toepassing van deze aanpak is wellicht bij LISP terug te vinden. De eerste formelere aanpak is van C. Strachey, die het onder andere gebruikte voor de definitie van een deel van de programmeertaal CPL. De methode is vervolmaakt door D. Scott.

De betekenis wordt vastgelegd door een functie F met twee argumenten, een programmatekst en een toestandsvector, en heeft als resultaat een toestandsvector: F: (Programma × Toestand) → Toestand.

Om de programmatekst visueel te scheiden van de toestandsvector, die als argument van F optreedt, wordt de programmatekst om-

sloten door het hakenpaar ⟦ en ⟧.

De betekenissen van de constructies, die ook bij de vorige methode als voorbeelden zijn gebruikt, worden nu beschreven door:

- assignment:
  F⟦x := e⟧(s) = s[x ← bereken(e, s)]
- statement list:
  F⟦st_1; st_2⟧(s) = (F⟦st_2⟧ . F⟦st_1⟧)(s)
  waarin (f.g)(x) = f(g(x)), dat wil zeggen dat de functie g wordt toegepast op x en dat op het resultaat hiervan f wordt toegepast.
- repetition:

```
F⟦while b do stl od⟧(s) =
      if bereken(b, s)
        then (F⟦while b do stl od⟧ . F⟦stl⟧)(s)
        else s
      fi
```

Het grote verschil met de vorige methode is het ontbreken van de rijen toestanden. Er wordt een volledig functionele beschrijving gegeven van de taal, waardoor de expliciete rij van toestanden uit de operationele methode nu impliciet is gegeven door de definitie van functionele compositie. Het verschil is bijvoorbeeld vooral van belang bij de repetitie, want bij de denotationele aanpak moet op een andere manier de inductie toegepast worden in het bewijs voor de correctheid van een programma.

De bovenstaande regels kunnen beschouwd worden als definities van functies die het effect van het programma(deel) bepalen.

De definities van deze functies zijn van de vorm

f = G(f)

waarin G een functionaal is (afbeelding van functies in functies). We kennen dit soort notaties bijvoorbeeld in de definitie van de faculteit:

fac(n) = <u>if</u> n = 0 <u>then</u> 1 <u>else</u> n * fac(n - 1).

We zullen hier niet verder op ingaan en het voorbeeld dan ook niet behandelen.

4.4 DE AXIOMATISCHE SEMANTIEK

De axiomatische semantiek maakt gebruik van de propositielogica. Deze wijze van semantiekdefinitie wordt dan ook wel de propositiesemantiek genoemd, maar in feite is de axiomatische aanpak slechts een van de methoden die gebruik maken van de propositielogica. Daar we ons tot deze beperken hebben we de bovenstaande naam gebruikt.

Men zou kunnen zeggen dat de denotationele semantiekdefinitie uit de operationele semantiekdefinitie wordt verkregen door de essentiële punten van de operationele definitie op een abstracte wijze te beschouwen. In de operationele methode wordt de betekenis gegeven door een regel die de rij toestanden geeft die het gevolg zijn van de uitvoering van een programma. In de denotationele methode worden de definities beschouwd als abstracte functies op toestanden zonder dat er sprake is van een rij. Is het wellicht mogelijk ook het begrip toestand uit de definities te verwijderen? Dit wordt gedaan bij de methode die we nu gaan bekijken.

De verzameling van objecten, de taal in termen waarvan de semantiek nu beschreven wordt, is de verzameling van formules uit een logisch systeem. Hoewel we deze formules kunnen zien als uitspraken over de toestanden, die in de vorige methoden een rol speelden, hoeven we deze formules niet op die manier te interpreteren.

De twee bekendste voorbeelden van de axiomatische semantiek zijn de methoden van R.W. Floyd en van C.A.R. Hoare. Bij Floyd's methode wordt uitgegaan van een stroomschema van het programma. Voor iedere verbinding in dit schema wordt een formule gegeven uit een logisch systeem (meestal de eerste orde predicatenlogica). Deze formules worden beweringen (assertions) genoemd. Tussen twee verbindingen zullen een of meer actiebeschrijvingen staan, waarvan de volgorde is vastgelegd door de paden in het stroomschema. Een pad tussen twee beweringen is geldig als de beweringen datgene karakteriseren wat op het pad gebeurt. Als deze twee beweringen behoren bij het begin en het einde van het programma en alle paden tussen deze beweringen zijn geldig, dan is het programma correct.

De methode van Hoare is gebaseerd op die van Floyd, er wordt echter geen stroomschema gebruikt. Bij Hoare legt de programmatekst de relatie vast tussen twee beweringen. In de methode van Hoare wordt de betekenis van een statement gedefinieerd door een relatie tussen twee beweringen (de bewering 'vooraf' en de bewering 'achteraf').

Als P en Q beweringen zijn en S is een programmaconstructie, dan is

{P} S {Q}

een notatie voor:
- Als P vooraf geldt, dan geldt na afloop (ten minste) Q; en:
- Als achteraf Q moet gelden, dan is het voldoende dat vooraf P geldt.

Zo schrijven we bijvoorbeeld:

{x = 3} x := x + 1 {x = 4}

Het axioma voor de *assignment* luidt:

$\{P^e_x\}$ x := e {P}

met als betekenis:

- Als na afloop de bewering P moet gelden, moet vooraf ook deze bewering gelden met voor iedere x de expressie e gesubstitueerd.

De betekenis van de *statement list* wordt vastgelegd door de volgende regel:

$$\frac{\{P\}\ st_1\ \{Q\},\ \{Q\}\ st_2\ \{R\}}{\{P\}\ st_1;\ st_2\ \{R\}}$$

Boven de streep in deze regel staan de voorwaarden die moeten gelden opdat de uitspraak, die onder streep staat, geldt.

Als de (juistheid van de) bewering P garandeert dat na afloop van st_1 de bewering Q geldt en als de (juistheid van de) bewering Q garandeert dat na afloop van st_2 de bewering R geldt, dan garandeert de (juistheid van de) bewering P dat na uitvoering van st_1; st_2 de bewering R geldt.

De betekenis van de *repetition*:

$$\frac{\{P \wedge B\}\ st\ \{P\}}{\{P\}\ \underline{\text{while}}\ B\ \underline{\text{do}}\ st\ \underline{\text{od}}\ \{P \wedge \neg B\}}$$

Deze regel wordt de *invariantiestelling* van Hoare genoemd.

Als het waar zijn van $P \wedge B$ garandeert dat de bewering P juist blijkt na uitvoering van st, dan wordt de juistheid van P gegarandeerd na afloop van de repetitie met st als statement (list) en B als conditie. Bovendien geldt dat B de waarde false heeft.

De regel doet geen uitspraak over de eindigheid van de repetitie.

De regel is alleen juist als de repetitie eindigt.

In de semantiek (en bij bewijsregels) gebruiken we bovendien dat, als $P \Rightarrow R$ en $S \Rightarrow Q$, dan geldt

$$\frac{\{R\}\ st\ \{S\}}{\{P\}\ st\ \{Q\}}$$

We gaan weer eens kijken naar het aantonen van de correctheid van het programma dat we ook bij de operationele methode gebruikt hebben:

```
z := 1; y := 0;
while y ≠ x do y := y + 1;
               z := z * y
            od
```

Wat we moeten bewijzen is dat

{true} programma {z = x!}

We kunnen eerst bewijzen dat geldt

{true} z := 1; y := 0 {z = y!}

en {z = y! ∧ y ≠ x} y := y + 1; z := z * y{z = y!}

Dit volgt uit de regels voor de assignment en de statement list.

Uit de regel voor de repetition vinden we dan

{z = y!} <u>while</u> y ≠ x <u>do</u> y := y + 1; z := z * y <u>od</u>{z = y! ∧ y = x}

zodat we krijgen

{true} programma {z = y! ∧ y = x}

(en (z = y! ∧ y = x) ⇒ z = x!).

Voor het aantonen van de correctheid moet ook nog de eindigheid aangetoond worden.

4.5 ENIGE OPMERKINGEN OVER DE METHODEN

Uit de behandeling van de drie methoden blijkt dat er steeds verder geabstraheerd wordt. In de overgang van de operationele naar de denotationele methode wordt geabstraheerd van het begrip rij (van toestanden). In de overgang van de denotationele naar de axiomatische aanpak wordt geabstraheerd van het begrip toestand.

Voor de voorbeelden (assignment, statement list en repetition) is dit verschil in abstractie reeds duidelijk; bij bijvoorbeeld procedures komt dit nog sterker tot uitdrukking. De operationele methode leidt dan tot een soort implementatiedetails die voor de programmeur niet van belang zijn, wel uiteraard voor de implementator. De denotationele aanpak blijkt vooral van nut voor de taalontwerper, de gewone gebruiker wordt bij het programmeren niet geholpen. Dit juist is het grote voordeel van de axiomatische methode: uit de semantiek krijgt de programmeur hints voor de toe te passen constructie (zie § 10.1).

5 VERTALEN

5.1 INLEIDING

De machine kent alleen de machinetaal. Gebruiken we een andere ('hogere') taal om onze programma's te schrijven, dan zullen deze programma's moeten worden vertaald naar de machinetaal. Dit gebeurt door een vertaalprogramma dat in de programmatuur van het systeem is opgenomen.

Elke elementaire handeling uit de 'hogere' taal moet dan door het vertaalprogramma worden omgezet in een aantal elementaire handelingen uit de machinetaal, zodanig dat het bedoelde en gedefinieerde effect wordt bereikt. De vertaling van alle elementaire handelingen, waaruit het programma in de 'hogere' taal is opgebouwd, levert een programma in machinetaal.

Een opdracht in de 'hogere' taal behoeft niet met één enkele opdracht in machinetaal te corresponderen.

Het programma in de 'hogere' taal noemt men *bronprogramma* (Engels: source program); het vertaalde programma heet *doelprogramma* (Engels: object program). De 'hogere' taal kan tot op zekere hoogte onafhankelijk van de machine worden gedefinieerd, waardoor een van de bezwaren tegen de machinetaal is ondervangen.

Het omzetten van een programma in een hogere taal naar een rij van opdrachten die kan worden uitgevoerd, kan ook geschieden met behulp van de interpreteertechniek. We komen hier in § 5.9 op terug.

Indien de 'bron'taal de symbolische machinetaal is (waarbij iedere instructie in de symbolische taal overeenkomt met één instructie in machinetaal), noemt men het vertaalprogramma een assembleerprogramma (Engels: assembler); is de gedefinieerde taal een hogere programmeertaal zoals Pascal, FORTRAN, COBOL of BASIC dan spreekt men van compileerprogramma of *vertaler* (Engels: *compiler*).

De complexiteit (denk aan enkele tienduizenden opdrachten) van het vertaalprogramma hangt uiteraard af van de complexiteit van de taal; de aard van het vertaalprogramma wordt in hoge mate bepaald door de structuur van de programmeertaal waarvoor ze is ontworpen.

De assembleerprogramma's voor symbolische machinetalen (even-

tueel met macrofaciliteiten) komen hier niet ter sprake; wij zullen de opbouw en de karakteristieken van vertalers voor hogere programmeertalen bekijken.

Bij een vertaler spelen drie talen een rol: de brontaal (de taal waarin de te vertalen programma's zijn geschreven), de doeltaal (de taal waarin het vertaalde programma is geschreven; is meestal de machinetaal), en de taal waarin de vertaler is geschreven (vaak dezelfde als de doeltaal, maar dat is niet nodig).

5.2 DEELACTIVITEITEN VAN EEN VERTALER

Het te vertalen programma, het bronprogramma, is een rij symbolen waaraan door de syntaxis een structuur wordt gegeven. Deze rij symbolen moet, door de vertaler en eventueel ook nog door andere delen van het operating system, uiteindelijk worden omgezet in een rij machine-instructies, het doelprogramma. In deze activiteiten kunnen een aantal deelactiviteiten worden onderscheiden:

- *Lexicografische analyse* (Engels: lexical analysis): het accepteren van de programmatekst als een rij symbolen, het herkennen van eenheden zoals getallen, identifiers, etc., en het genereren van een rij tokens (zie § 5.3) die elk een programma-eenheid als enkelvoudig item representeren. De rij tokens zal, als geheel, de programmastructuur weerspiegelen. Het deel van de vertaler dat deze taken uitvoert noemt men wel de scanner.
- Het bijhouden van een *lijst* met namen (Engels: *symbol table*) die bijvoorbeeld gebruikt wordt om na te gaan of alle gebruikte namen van identifiers zijn gedeclareerd, en waarin bijvoorbeeld bij uitvoering de waarden (of de toegewezen adressen) van de identifiers kunnen worden opgeslagen.
- *Syntactische analyse* (Engels: syntax analysis, parsing): het herkennen van de syntactische structuur van het programma door middel van de rij tokens en het tegelijkertijd genereren van de afleidingsboom.
- *Semantische analyse* (Engels: semantic analysis): het controleren van de context-gevoelige regels van de programmeertaal, bijvoorbeeld: gebruikte identifiers moeten gedeclareerd zijn, typecontrole, etc.
- Het vinden van *fouten* en het genereren van foutmeldingen.
- Het vertalen van de afleidingsboom naar een equivalent programma in de doeltaal: *codegeneratie*.
- Het *optimaliseren* van het doelprogramma wat betreft geheugengebruik en/of executietijd.
- Het aan het programma *koppelen* van (afzonderlijk vertaalde) programmadelen (bijvoorbeeld routines) die bij uitvoering van het

programma nodig zijn, en het *gereedmaken* van het programma voor verwerking. Deze taken worden respectievelijk 'linking' en 'loading' genoemd en behoren niet tot de eigenlijke vertaaltaak.

De hierboven genoemde taken worden niet in iedere vertaler in dezelfde volgorde verricht. Vaak worden de taken ook gecombineerd uitgevoerd omdat ze sterk afhankelijk van elkaar zijn: denk bijvoorbeeld aan de syntactische analyse en het vinden van fouten. Als tijdens het vertaalproces het beschikbare geheugen klein is, zal het nuttig zijn de activiteiten te scheiden om niet te veel geheugen te verliezen voor de code van de vertaler; in dit geval zal de vertaler meerdere malen de programmatekst, zij het in verschillende vormen (symbolen, tokens, boom, etc.), doorlopen; het aantal malen dat dit gebeurt noemt men het aantal 'passes' van de vertaler. Indien het mogelijk is alle activiteiten te combineren zodat slechts één 'pass' nodig is, zullen de 'interne' representaties van het programma niet nodig zijn; niet alle talen zijn echter van een zodanige structuur dat één-pass vertaling mogelijk is.

Het is echter altijd aan te raden bij het maken van een vertaler de taken zoveel mogelijk gescheiden te houden en in afzonderlijke programmadelen onder te brengen, opdat de vertaler overzichtelijk blijft. In de rest van dit hoofdstuk zullen we de taken afzonderlijk bespreken alsof we een meer-pass vertaler behandelen.

5.3 LEXICOGRAFISCHE ANALYSE

De *scanner* neemt het bronprogramma teken voor teken door en probeert in de tekst speciale symbolen (begin, if, while, etc.), identifiers en numbers te herkennen. De scanner maakt van deze eenheden enkelvoudige symbolen (Engels: *tokens*) en geeft dan het programma als een rij van tokens door aan de parser die de verdere syntactische analyse verzorgt.

De taak van de scanner is dus tweeledig:

- het herkennen van de programma-eenheden; dit is een vorm van syntactische analyse;
- het omzetten van het programma als een rij symbolen naar een rij tokens; dit is een vorm van vertalen.

Dat de scanner afzonderlijk wordt gezien heeft een aantal redenen, waarvan we hier noemen:

- de syntactische analyse van het programma is eenvoudiger als met grotere eenheden (tokens) kan worden gewerkt;
- men kan de eigenlijke vertaling onafhankelijk maken van de programmatekst op een gegevensdrager.

De programma-eenheden voor de in § 3.3 gedefinieerde eenvoudige programmeertaal kunnen worden onderscheiden in vijf klassen:

- identifiers
- numbers
- reserved words
- operatoren (+, -, *, /, <, ≠, etc.)
- speciale tekens (:=, ;, (en), etc.).

Er zullen dus ook vijf klassen tokens gegenereerd worden. Een token bevat twee soorten informatie:
- de klasse van de eenheid (deze informatie wordt door de parser gebruikt), en
- de aanduiding van het specifieke element van de klasse, bijvoorbeeld de waarde, de index in de symbol table, etc. (deze informatie wordt gebruikt bij semantische analyse en codegeneratie).

VOORBEELD
De constructie A := B + 3; bevat zes eenheden: de namen A en B, de constante 3, de assignment operator :=, de aritmetische operator + en het speciale teken ;. Hiervoor moeten dus zes tokens gegenereerd worden.
(einde voorbeeld)

De structuren van deze eenheden kunnen beschreven worden door middel van reguliere grammatica's.

De scanner kan steeds door het eerste karakter van een eenheid bepalen in welke klasse die thuishoort:
- identifiers beginnen met een letter
- numbers beginnen met een cijfer, een punt of een $_{10}$
- operatoren zijn herkenbaar
- reserved words zijn onderstreept, dus de eerste letter is als zodanig herkenbaar
- de speciale symbolen zijn ook herkenbaar.

Het eerstvolgende te verwerken karakter bepaalt dus steeds volgens welke regels de volgende eenheid herkend en verwerkt moet worden. We kunnen de scanner dus globaal als volgt beschrijven:

```
getnextchar(x);
while 'x ≠ $' do
     case x of letter        : 'scan identifier';
               digit, ., 10  : 'scan number';
               underscore    : 'scan reserved word';
               operator      : 'scan operator';
               special       : 'scan special symbol';
               else          : fout := true
     end;
     if fout then ... fi
          od
```

In de verschillende scan-delen gebeurt het herkennen van de eenheid en tevens het verzorgen van de tokens. Nadat een eenheid

geheel herkend is, moet dus soms de symbol table uitgebreid of geraadpleegd worden en de juiste informatie aan de tokens worden toegevoegd. In geval een fout binnen een eenheid gedetecteerd is, moet dat geregistreerd worden: een foutmelding wordt gegenereerd en het startpunt van de volgende eenheid wordt bepaald.

De scanner herkent steeds zo groot mogelijke eenheden, dat wil zeggen de symboolrij wordt zo ver mogelijk herkend, tot een symbool gezien wordt dat zeker niet bij deze eenheid behoort.

In hoofdstuk 3 bespraken we de reguliere grammatica voor numbers, een tabel om een karakter als number te analyseren en een automaat die numbers accepteert. Hiermee kan op eenvoudige wijze een herkenningsalgoritme afgeleid worden. We kunnen in dit geval ook direct de waarde van de constante berekenen en deze als informatie aan het te genereren token toevoegen. Kortom, we bepalen direct de semantische inhoud van een number. We doen dit door bij sommige toestandsovergangen de juiste actie te specificeren:
we berekenen de waarde in g en maken daarbij gebruik van
- k: geeft het rangnummer van de te verwerken decimaal aan
- e: berekent de waarde van de exponent.

Tevens voegen we de herkenning van de syntactische fouten toe.

We geven eerst de uitgebreide tabel.

Vanuit toestand	met karakter	actie	naar toestand
1	d		2
	.		3
	$_{10}$		5
2	d	g := g*10+x	2
	.		3
	$_{10}$		5
3	d	k := k+1; g := g+x*(10↑-k)	4
	anders		fout
4	d	k := k+1; g := g+x*(10↑-k)	4
	$_{10}$		5
5	+/-	s := s+1 / s := s-1	6
	d	e := e*10+x	7
	anders		fout
6	d	e := e*10+x	7
	anders		fout
7	d	e := e*10+x	7

Het programmadeel 'scan number' wordt dan:

```
g := 0; fout := false;
while 'x=digit' do g := 10*g+x;
                   getnextchar(x)
                od;
if 'x=.' then k := 0;
              getnextchar(x);
              if 'x≠digit' then fout := true fi;
              while 'x=digit' do k := k+1;
                                 g := g+x*(10↑(-k));
                                 getnextchar(x)
                              od
fi;
if 'x=₁₀' ∧ ¬fout then e := 0;
                       getnextchar(x);
                       if 'x=-' then s := -1; getnextchar(x)
                                else if 'x=+' then s := +1;
                                                   getnextchar(x)
                       fi fi;
                       if 'x≠digit' then fout := true fi;
                       while 'x=digit' do e := e*10+x;
                                          getnextchar(x)
                                       od;
                       g := g*(10↑(s*e))
fi;
'genereer token van klasse 'number' en verwerk dat token'
```

OPMERKING
De scanner kan het eerste zelfstandige deel van de vertaler zijn dat het programma in de bewerkte brontaal in de één of andere vorm in zijn geheel aan de andere delen van de vertaler overdraagt. Het kan ook zijn dat de scanner als procedure vanuit de parser wordt geactiveerd.

5.4 SYNTACTISCHE ANALYSE

De *parser* moet de structuur van een programma herkennen. Dit gebeurt aan de hand van de syntaxis-regels van de taal. De meeste programmeertalen worden beschreven met context-vrije grammatica's; daarom beschouwen we hier alleen het parsen van zinnen uit een context-vrije grammatica.

In hoofdstuk 3 zagen we reeds dat context-vrije grammatica's geaccepteerd worden door non-deterministische push-down automaten en reguliere grammatica's door deterministische eindige automa-

ten. Het analyseren van zinnen uit contact-vrije talen zal dus minder eenvoudig zijn dan het analyseren van zinnen uit een reguliere taal.

De parser werkt met een rij tokens die herkend moet worden als een door de grammatica toegestane programmastructuur. Dat gebeurt door de desbetreffende syntactische boom te genereren; deze boom wordt dan later gebruikt voor de codegeneratie (evenals in hoofdstuk 3 stellen we ons bomen voor met de wortel boven en de bladeren onder). De parser houdt zich dus bezig met het probleem: is van deze rij tokens de syntactische boom te bepalen? Zo ja, genereer die boom, zo neen, maak dat kenbaar. We beschouwen alleen grammatica's die niet-ambigu zijn, zodat van dé syntactische boom sprake kan zijn.

Parsing-algoritmen kunnen in twee klassen worden ingedeeld:

- de *top-down* algoritmen bouwen de syntactische boom op door bij de wortel (het startsymbool) te beginnen om uiteindelijk in de bladeren van de boom de te herkennen zin te vinden, en
- de *bottom-up* methoden bouwen de boom vanuit de bladeren op door steeds te reduceren (het rechterdeel van een produktieregel vervangen door het linkerdeel) en uiteindelijk bij het startsymbool uit te komen.

In het volgende beschouwen we alleen van-links-naar-rechts werkende parsing methoden. Een top-down algoritme heet LL, als van Links naar rechts de karakters van de te herkennen zin verwerkt worden en als de afleidingsboom opgebouwd wordt door een Linkse afleiding te bepalen. Een bottom-up algoritme heet LR, als van Links naar rechts de karakters van de te herkennen zin verwerkt worden en als de reductie steeds de meest linkse reduceerbare rij karakters reduceert; een rij heet reduceerbaar als die voorkomt als rechterlid van een produktieregel; merk op dat hierbij de inverse van een Rechtse afleiding bepaald wordt.

VOORBEELD

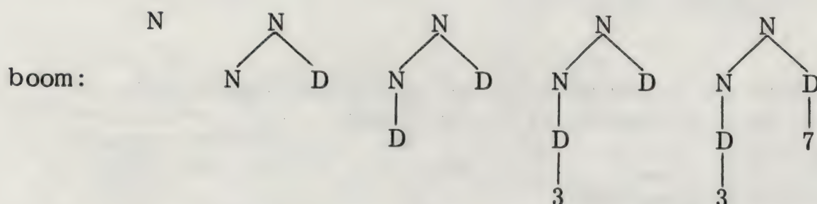
Beschouw de volgende produktieregels:

N ::= D|ND

D ::= 0|1|2|3|4|5|6|7|8|9

We parsen de 'zin' 37, kortom we zoeken een afleiding en construeren tegelijkertijd de syntactische boom.

Allereerst top-down:

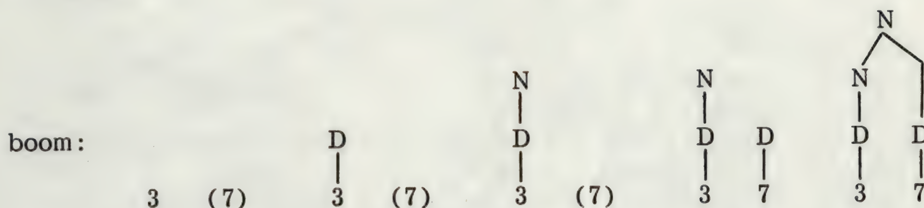


Afleiding:

N ⇒ ND ⇒ DD ⇒ 3D ⇒ 37

(de karakters worden 'gelezen' van links naar rechts: eerst de 3 daarna de 7)

Nu bottom-up:



Afleiding: 37 ⇐ D7 ⇐ N7 ⇐ ND ⇐ N
(ook hier worden de karakters van links naar rechts 'gelezen')
(einde voorbeeld)

In het voorbeeld toonden we alleen de goede afleidingen/reducties. Het is echter niet zo eenvoudig steeds de goede voortzetting te vinden. De belangrijkste problemen bij parsing zijn:

- als bij een top-down methode een nonterminal A vervangen moet worden door een rechterlid van één van de produktieregels:
  A ::= x1
  A ::= x2
  .
  .
  A ::= xn
  Welke regel moet dan gekozen worden?
- hoe vinden we bij bottom-up de meest linkse, reduceerbare rij symbolen en tot welke nonterminal moet die gereduceerd worden?

Een mogelijke oplossing voor die problemen is steeds één van de mogelijke alternatieven willekeurig te kiezen en maar te proberen. Als later blijkt dat deze mogelijkheid niets oplevert, wordt het volgende alternatief geprobeerd. Het zal duidelijk zijn dat deze backtrack methode erg veel werk en erg veel administratie vergt.

Voor zowel top-down als bottom-up methoden is winst te behalen door de context van de te parsen deelrij te beschouwen of door de structuur van produktieregels te beperken. In het volgende gaan we kort op top-down en bottom-up methoden in.

Top-down parsing

We beschrijven eerst globaal het backtracking algoritme.

- Nummer voor elke nonterminal A de verschillende rechterdelen van produktieregels waar A linkerdeel is; alle produktieregels met A als linkerdeel kunnen dan voorgesteld worden door
  A ::= $\alpha_1 | \alpha_2 | \ldots | \alpha_n$
- Begin met een boom bestaande uit één knoop met als label het startsymbool S. Initieel is dat de 'actieve' knoop. Het eerste symbool van de te herkennen zin wordt 'current symbol'.

- Voer recursief de volgende stappen uit.
  1. Als de actieve knoop als label een nonterminal heeft, zeg A, kies dan het eerste alternatief voor A, zeg $X_1X_2...X_k$ en creëer k directe opvolgers van A, met labels X_1, X_2, ..., X_k. De knoop met X_1 wordt actief. Als k=0 ($A \rightarrow \varepsilon$) maak dan de knoop rechts van A actief.
  2. Als de actieve knoop als label een terminal heeft, zeg a, vergelijk dan het 'current symbol' met a.
     Als ze overeenkomen wordt het eerstvolgende symbool van de zin 'current symbol'; de (meest linkse) knoop rechts van a wordt actief.
     Komen ze niet overeen dan wordt de knoop, waar de laatste produktieregel was toegepast, actief en wordt, nadat de waarde van 'current symbol' hersteld is, het volgende alternatief geprobeerd. Als geen alternatief meer aanwezig is, ga dan terug naar de knoop waar de op een na laatste produktieregel was toegepast en probeer daar het volgende alternatief, etc.

Dit algoritme kan geïmplementeerd worden door een parser die gebruik maakt van een set recursieve procedures, een zogenaamde *recursive descent* parser. Voor elke nonterminal A is er een recursieve procedure, die de alternatieven voor afleidingen van A onderzoekt, met als parameters de nog te herkennen symboolrij (die begint met het 'current symbol') en een boolean die aangeeft of de afleiding gevongen is. De procedure voor A, genaamd A, zal andere procedures aanroepen om een afleiding van A te vinden. We herinneren eraan dat de rij te herkennen symbolen in feite een rij tokens is.

VOORBEELD
Voor de produktieregels voor de eenvoudige programmeertaal uit § 3.3 zullen de procedures voor de nonterminals <program> en <exp2> er uit zien als weergegeven op p.71.
(einde voorbeeld)

We hebben in het voorbeeld de procedures erg eenvoudig voorgesteld. We noemen nu nog enkele zaken die van belang zijn.

1. Na aanroep van de procedure program zal de waarde van de actuele boolean parameter aangeven of de initiële rij tokens een zin uit de taal was ja dan nee. Enige informatie over de aard van eventuele fouten wordt niet verstrekt. Het is echter zeer wel mogelijk de procedures te wijzigen zo dat foutmeldingen gegenereerd worden. Ook is het mogelijk de procedures te wijzigen zo dat na detectie van een fout toch de rest van de rij geanalyseerd wordt; daartoe wordt dan ofwel een deel van de rij tokens overgeslagen en vervolgd bij bijvoorbeeld het volgende speciale teken, ofwel een rij tokens tussengevoegd zodat wel een afleiding te maken was; van beide acties wordt dan – als het goed is – wel melding gemaakt. Detectie en herstel van fouten is een erg gecompliceerde zaak, waar we hier niet verder op zullen ingaan.

```
procedure program (var S : sequence of token, var b : boolean);
begin var t : token;
      block(S, b);
      if b∧S≠() then t from S;
                     if 't≠$' then b := false fi
      fi
end;

procedure exp2 (var S : sequence of token, var b : boolean);
begin var t : token; var Shulp : sequence of token;
      if S = () then b := false
      else t from S;
           if 't=input' then b := true
           else t backto S; Shulp := S;
                identifier(S, b);
                if ¬b then S := Shulp; number(S, b);
                   if ¬b then S := Shulp; t from S;
                      if 't=(' then expression(S, b);
                         if b∧S≠() then t from S;
                            if 't≠)' then S := Shulp;
                               b := false
                            fi
                            else b := false; S := Shulp
                         fi
                      fi
                   fi
                fi
           fi
      fi
end
```

2. Het genereren van de syntactische boom kan tijdens de afleiding door de procedures verzorgd worden. Nadat een representatie voor de boom gekozen is, kan elke procedure aangepast worden. Omdat dit erg machine-afhankelijk is (ook in verband met codegeneratie), gaan we daar hier niet verder op in.
3. De semantische analyse kan ook aan elke procedure toegevoegd worden. Het is dan niet nodig deze analyse na de syntactische uit te voeren, maar dat gebeurt als het ware tegelijkertijd.
4. Parsen volgens de recursive descent methode is een simulatie van een push-down automaat. De stack die bij de automaat gebruikt wordt, wordt hier gerealiseerd door het procedure-mechanisme.

Er zijn een aantal problemen bij deze aanpak. Het eerste betreft links-recursieve produktieregels, dat zijn regels van de vorm $A \overset{*}{\Rightarrow} Aa$. Deze links-recursie kan bij de recursive descent parser leiden tot een oneindig vaak aanroepen van de procedure A. Het is eenvoudig

de produktieregels te wijzigen zodat directe links-recursie niet meer voorkomt.

VOORBEELD
Als een produktieregel is: $A \rightarrow A\alpha | \beta$
(waarin β A niet bevat)
vervangen we deze door: $A \rightarrow \beta A'$
$A' \rightarrow \alpha A' | \varepsilon$
(einde voorbeeld)

Het is echter moeilijker de produktieregels te wijzigen zodat ook indirecte links-recursie niet meer voorkomt. Daarom wordt top-down analyse meestal beperkt tot grammatica's zonder links-recursie.

Een tweede probleem betreft de toevoeging van de semantische analyse aan de backtracking methode. Indien bij het construeren van de afleidingsboom een subboom vernietigd moet worden omdat deze geen correcte voortzetting leverde, moet ook al het werk in verband met de semantische analyse ongedaan gemaakt worden; bijvoorbeeld: entries in de symbol table moeten verwijderd worden. Aangezien dat veel overhead met zich meebrengt, is het aantrekkelijk geen backtracking te gebruiken.

Het derde probleem is de efficiëntie van de backtracking methode. Het is een soort trial-and-error methode, en dus erg tijdrovend.

Een oplossing voor de laatste twee problemen is bij de analyse steeds gebruik te maken van de k volgende symbolen ($k \geq 1$) van de nog af te leiden zin. De parser wordt deterministisch als uit deze k symbolen steeds voldoende informatie te halen is om de juiste keuze te maken. Het spreekt voor zich dat dit ook eisen stelt aan de grammatica; een grammatica die zich voor zo'n top-down parser leent, noemt men *LL(k)*: de parser werkt van-Links-naar-rechts, produceert een Linkse afleiding en kijkt k symbolen vooruit.

DEFINITIE
Een grammatica G is LL(1) dan en slechts dan als:
indien $A \rightarrow \alpha$ en $A \rightarrow \beta$ twee verschillende produktieregels zijn, moet het volgende gelden:
$\text{eersten}(\alpha) \cap \text{eersten}(\beta) = \emptyset$ met voor $\gamma \in V^*$:
$\text{eersten}(\gamma) = \{a \in T \mid \text{er bestaat een afleiding } \gamma \overset{*}{\Rightarrow} a\delta \text{ met } \delta \in V^*\}$.
(einde definitie)

Deze definitie geldt mits de regel $A \rightarrow \varepsilon$ niet voorkomt. Indien dat wel het geval is moet de eis sterker worden (zie literatuur).

Dit alles komt neer op de eis dat uit het eerstvolgende symbool σ te concluderen is welke produktieregel gekozen moet worden: als $\sigma \in \text{eersten}(\alpha)$ dan $A \rightarrow \alpha$, als $\sigma \in \text{eersten}(\beta)$ dan $A \rightarrow \beta$.

Er zijn vergelijkbare eisen opdat een grammatica LL(K) is, $k \geq 2$.

De klasse der LL(k)-grammatica's, k ≧ 1, is een deelverzameling der context-vrije grammatica's. Voor elke LL(k)-grammatica is een deterministische parser te maken en ook voor reguliere grammatica's is dat mogelijk (zie § 5.3). Er geldt: de klasse der reguliere talen is een echte deelverzameling van de klasse der LL(1)-talen, deze is een echte deelverzameling van de klasse der deterministische context-vrije talen en deze is een echte deelverzameling van de gehele verzameling van context-vrije talen.

Voor een verdere analyse van top-down parsing verwijzen we naar de literatuur.

Bottom-up parsing

Het proces dat we hier beschrijven probeert, in omgekeerde volgorde, een rechtse afleiding van een zin te vinden door alle mogelijke reducties op de zin toe te passen. De zin wordt van links naar rechts doorlopen. Deze vorm heet '*shift-reduce* parsing'. We beschrijven eerst de backtracking methode. Een stap van de parser bestaat uit het inspecteren van de symbolen op de top van de stapel om te zien of een rechterdeel van een produktieregel hiermee overeenkomt. Als dat zo is, wordt een reductie toegepast door de symbolen te vervangen door de nonterminal die het linkerdeel van die produktieregel is; als meerdere reducties mogelijk zijn, worden de alternatieven genummerd en één voor één geprobeerd. Als geen reductie mogelijk is, wordt het volgende symbool van de te herkennen zin op de stapel gezet en wordt vervolgd. Altijd wordt geprobeerd te reduceren alvorens het volgende symbool op de stack te zetten. Als het einde van de te herkennen zin is bereikt en nog geen reductie mogelijk is, wordt de laatst gerealiseerde reductie ongedaan gemaakt en wordt daar het volgende alternatief geprobeerd; als er geen volgend alternatief is, wordt de voorlaatst gerealiseerde reductie ongedaan gemaakt, enz.

VOORBEELD

produktieregels : S → AB
A → ab
B → aba

te herkennen zin: ababa

Eerst wordt a op de stack gezet. Omdat geen reductie mogelijk is, wordt b op de stack gezet. Dan worden b en a van de stack gehaald en A erop gezet. Omdat geen reductie voor A mogelijk is, wordt a op de stack gezet en vervolgens om dezelfde reden b. Ook nu worden b en a van de stack gehaald en A erop gezet. Geen reductie is mogelijk en dus wordt a op de stack gezet. Nu is geen verdere reductie mogelijk.

Nu wordt teruggegaan naar het punt waar de laatste reductie uitgevoerd werd, dat is toen Aab (b op de top) op de stack stond. Omdat geen andere reductie mogelijk is, wordt nu a op de stack gezet: deze bevat nu Aaba. Nu is aba reduceerbaar tot B en dus zal

de stack dan AB bevatten. Dit is te reduceren tot S en dus is het parsing proces afgelopen; de rij ababa is correct.
(einde voorbeeld)

Een probleem bij bottom-up parsing is het voorkomen van cycles, bijvoorbeeld afleidingen van de vorm A $\overset{*}{\Rightarrow}$ A; dit kan leiden tot oneindige repetities; we gaan er verder van uit dat de grammatica geen cycles bevat. Ook afleidingen van de vorm A $\overset{*}{\Rightarrow}$ ε zijn een probleem omdat altijd een reductie mogelijk is: de lege string boven op de stapel reduceren tot een nonterminal. Dit probleem is oplosbaar, maar in het volgende verbieden we produktieregels van de vorm A $\rightarrow$ ε.

Een ander probleem is de efficiëntie. Vaak zijn verschillende reducties mogelijk en zal deze trial-and-error methode erg tijdrovend zijn. Een aanzienlijke verbetering ontstaat als direct de goede keuze gemaakt kan worden. Ook is het moeilijk de meest linkse reduceerbare rij symbolen te bepalen, uitgaande van een rij symbolen die ontstaan is na een aantal reducties. Als dit zoekproces te versnellen is zodat direct de meest linkse reduceerbare rij symbolen gekozen kan worden, zal dat de efficiëntie wederom verbeteren. We bespreken een methode om de bottom-up methode deterministisch te maken.

LR(k)-grammatica's
Als bij de analyse gebruik gemaakt kan worden van de k volgende symbolen (k $\geq$ 1) van de te herkennen zin, is voor sommige grammatica's eenduidig te bepalen welke reductie moet worden uitgevoerd. Zulke grammatica's heten LR(k): van-Links-naar-rechts werkend, een Rechtse afleiding producerend en k symbolen vooruitkijkend. De eis dat er een grammatica LR(k) is komt er, informeel gesteld, op neer dat inspectie van de inhoud van de stack en van de k eerstvolgende symbolen van de te herkennen zin altijd uitsluitsel geeft of de rij symbolen op de stack (terminals en nonterminals) gereduceerd moet worden of een volgend symbool van de te herkennen zin op de stack geplaatst moet worden. Kortom, voor LR(k)-grammatica's zijn efficiënte parsers te maken.

Een eigenschap: Elke taal waarvan de zinnen LL(k) of LR(k) te ontleden zijn, is ook LR(1) te ontleden.

Het blijft erg ingewikkeld een LR-parser te maken. Vaak wordt hiervoor een 'parser generator' gebruikt, een hulpprogramma dat, gegeven een LR(k)-grammatica, een parser produceert. Voor elke LR(k)-grammatica kan een deterministische parser gemaakt worden. Omdat de meeste programmeertaal-constructies zijn te genereren met LR(k)-grammatica's, zijn reeds vele 'parser-generators' gerealiseerd. Voor een verdere analyse van LR-parsing verwijzen we wederom naar de literatuur.

5.5 SEMANTISCHE ANALYSE

In deze fase van het vertaalproces wordt de programmatekst, gerepresenteerd in de afleidingsboom, gecontroleerd aan de hand van de *context-gevoelige* regels van de taal. De context-gevoelige regels zijn minder geformaliseerd dan de context-vrije; voor de verwerking van de context-gevoelige regels zijn dan ook geen algemeen aanvaarde methoden en technieken beschikbaar. We beschrijven hier een drietal onderwerpen.

1. Identifiers moeten gedeclareerd zijn alvorens ze gebruikt kunnen worden. We zagen reeds dat de vertaler een *symbol table* bijhoudt waarin de gedeclareerde identifiers worden gerepresenteerd. Indien in het programma een niet-gedeclareerde variabele wordt gebruikt, kan dat geconstateerd worden met behulp van de symbol table; deze fout zal aanleiding geven tot een foutmelding. Indien de vertaler deze niet-gedeclareerde variabele nu toevoegt aan de symbol table, bijvoorbeeld als zijnde van het type 'onbepaald', worden verdere foutmeldingen betreffende deze variabele onderdrukt. Als de symbol table niet aangepast wordt, kan het voorkomen dat een groot aantal identieke foutmeldingen wordt gegenereerd, die in feite alle slechts op één fout betrekking hebben.
   Bij blok-gestructureerde programmeertalen zal tijdens de analyse de symbol table groeien en slinken. Indien de symbol table ook nog wordt gebruikt tijdens codegeneratie en/of executie, zal de symbol table tijdens de analyse alleen groeien, maar zullen steeds slechts stukken ervan (in analogie van de blokken) gebruikt worden voor de controle.
2. Van expressies moet het type bepaald worden; dat type wordt gebruikt bij:
   - de controle of het een semantisch correcte expressie is, bijvoorbeeld: worden niet de waarden van twee boolean variabelen opgeteld?
   - in een assignment statement wordt de waarde van de expressie toegekend aan de variabele; zijn die wel van hetzelfde type?
   - als de machine voor sommige operaties meerdere instructies kent, bijvoorbeeld vermenigvuldiging van twee integers en vermenigvuldiging van twee reals, is het voor de codegeneratie van belang welk type operatie gewenst is.

   Ook hier kan het type 'onbepaald' overbodige foutmeldingen voorkomen.
3. De evaluatie van expressies verloopt volgens gespecificeerde prioriteitsregels. Het genereren van code voor de berekening van de waarde van een – zeg aritmetische – expressie, zal van die prioriteitsregels afhankelijk zijn. Voor de gebruikelijke *infix*-notatie (operator tussen operanden) is dit erg moeilijk. Bij-

voorbeeld: in de expressie A+(B*C) mag de optelling pas uitgevoerd worden als de waarde van de tweede operand (B*C) bekend is; de code voor berekening van de tweede operand moet dus voorafgaan aan de optel-instructie.

Een aantrekkelijker notatie is *postfix*: de operator volgt op de operanden. Dus de infix-expressie A+B wordt AB+ in postfix en A+(B*C) wordt genoteerd als ABC*+. In de postfix-notatie komen geen haakjes voor en de operanden gaan vooraf aan de operator. Het genereren van code voor een postfix-expressie levert weinig problemen op. Het omzetten van infix-expressies naar postfix-expressies volgens de heersende prioriteitsregels (semantiek) is dus een zinvolle taak, die als eerste fase van de codegeneratie beschouwd kan worden.

5.6 GENEREREN VAN CODE

Bij de *codegeneratie* wordt het programma zoals dat gerepresenteerd is in de afleidingsboom, vertaald naar een programma in de doeltaal (code). De eerste fase van dit proces, het omzetten van infix-expressies naar postfix-expressies, is reeds genoemd in § 5.5.

Niet iedere vertaler genereert dezelfde soort *code*. Men kan in dit opzicht drie soorten vertalers onderscheiden.

De code die wordt gegenereerd kan machinetaal zijn met absolute adressering (Engels: core image code). Het programma wordt vertaald en direct verwerkt. Dit kan alleen als het programma geheel zelfstandig is en geen gebruik maakt van andere (al vertaalde) programmadelen. Vooral voor kleine programma's die slechts éénmaal worden gebruikt kan een vertaler, die op deze wijze werkt, goed worden gebruikt. Men krijgt op deze wijze een tijdwinst, die aanzienlijk kan zijn.

Een andere methode is het genereren van symbolische machinetaal. Dit is een gemakkelijker wijze van werken, omdat men niet tot het niveau van de bits behoeft te gaan. Het grote nadeel is echter, dat men later nog een vertaalslag naar machinetaal moet maken door deze tekst aan het assembleerprogramma aan te bieden.

De derde, en meest gebruikte, methode is het genereren van doeltaalcode, dat wil zeggen machinetaal met openlating van de adressen. Het gegenereerde machinetaal-programma bevat ook nog namen of aanduidingen van programmadelen die in het programma moeten worden gebruikt of die van dit programma gebruikmaken. Na de vertaling zal het programma in een bibliotheek worden gezet. De linkage editor en het laadprogramma zorgen ervoor dat het programma (eventueel later) de gedaante krijgt waarin het kan worden verwerkt.

5.7 HULP DIE DE VERTALER (EN HET SYSTEEM) BIEDT BIJ NIET-CORRECTE PROGRAMMA'S

In alle fasen van de compilatie kan de vertaler ontdekken, dat de aangeboden (bron)tekst geen correct programma is. Bij het signaleren van een *fout* kan de vertaler direct stoppen, omdat de rest van het programma zinloos is. De vertaler zal echter vaak trachten ook de rest van het programma de bekijken (zonder dat er codegeneratie plaatsvindt), om de programmeur te waarschuwen als er nog meer fouten in het programma zitten. Het is natuurlijk mogelijk dat de vertaler fouten aangeeft die zijn ontstaan doordat een bepaald stuk programma niet te interpreteren was door een eerdere fout.

De vertaler zal dus nagaan of het programma syntactisch en semantisch correct is (statische controle). Sommige fouten kunnen echter pas tijdens de uitvoering van het programma worden geconstateerd (dynamische controle); als we als index in een array bijvoorbeeld een variabele gebruiken, zal pas tijdens uitvoering kunnen worden vastgesteld of de waarde van deze variabele binnen de gedeclareerde grenzen blijft.

De programmeur heeft uiteraard weinig aan de melding 'PROGRAMMA IS FOUT'; hij zal graag willen weten waar de fout is opgetreden en van welke aard de fout is. Een goed systeem en de vertaler zullen deze informatie dan ook verstrekken door de plaats in de programmatekst aan te geven waar de fout is opgetreden en (bij dynamische fouten) de waarden van de relevante variabelen op te geven.

Als het programma nu syntactisch en semantisch correct is en geen dynamische fouten veroorzaakte, wil dat nog niet zeggen, dat het ook goed is; het programma moet ook pragmatisch juist zijn. Als wij, om het heel eenvoudig te stellen, twee variabelen in een programma willen vermenigvuldigen en wij schrijven een plusteken, kan geen vertaler deze fout constateren. Om zich tegen dit soort fouten te wapenen zal een programmeur zeer zorgvuldig te werk moeten gaan en de correctheid van zijn programma op voorhand moeten bewijzen.

5.8 DE VERTALER ALS ONDERDEEL VAN HET OPERATING SYSTEM

We zullen het operating system bespreken voorzover er een directe samenhang is met het vertalen.

We kunnen het operating system beschrijven als de programmatuur, die, gebruikmakend van de apparatuur, in staat is programma's uit te voeren, er daarbij voor zorgdragend dat in die programma's volledig gebruik kan worden gemaakt van alle faciliteiten

van die apparatuur (randapparatuur, geheugen, centrale verwerkingseenheid) en alle door het systeem zelf gecreëerde faciliteiten van de programmatuur (vertalers, bestanden, bibliotheek).

Het operating system bevat in de meeste gevallen de vertaler als routine; het is dan mogelijk meer vertalers in het systeem op te nemen, waardoor de machine met hetzelfde systeem in staat is programma's, geschreven in verschillende talen, te verwerken. Het is ook mogelijk dat voor één taal een aantal vertalers aanwezig is; dit kan bijvoorbeeld als men programma's moet vertalen, waarvan men weet dat ze maar van een beperkte subset van de taal gebruik maken, of als men weet dat de programma's slechts eenmalig worden verwerkt, zodat men bijvoorbeeld de optimalisering kan weglaten.

De van programmatuur voorziene computer is de machine waar de gebruiker mee te maken heeft. De gebruiker behoeft zich 'slechts' de specificaties van de 'aangeklede' computer eigen te maken, deze specificaties bestaan voor een groot deel uit specificaties van de gebruikte taal.

De omzetting van brontaal naar een door de machine verwerkbare code is de taak van drie standaardsysteemprogramma's, namelijk een vertaler (of assembleerprogramma), de *linkage editor* en het *laadprogramma*.

De vertaler (of het assembleerprogramma) zet de bronmodule om in een zogenaamde doelmodule; deze doelmodule bevat het doelprogramma, dit is de procesrepresentatie in een nog niet uitvoerbare vorm, en een hoeveelheid informatie voor de linkage editor.

De linkage editor zet een aantal doelmodulen om in een laadmodule (Engels: core image module); deze laadmodule bevat de zogenaamde core image code, dit is de procesrepresentatie in uitvoerbare vorm, benevens een hoeveelheid informatie voor het laadprogramma. We hebben nu onderscheid gemaakt tussen het resultaat van een vertaling in doeltaal en een uiteindelijk laadbaar programma (in core image code). In een doelprogramma kunnen nog niet vertaalde verwijzingen voorkomen naar andere programmadelen (modulen), procedures en macro's.

Het laadprogramma plaatst het programma in core image code in het virtuele geheugen waarbij gebruik gemaakt wordt van de informatie in de laadmodule (bijvoorbeeld de omvang van het programma). Nadat het laadprogramma dit werk heeft beëindigd, bevindt zich in het virtuele geheugen een direct uitvoerbare procesrepresentatie.

Wij zullen in het navolgende achtereenvolgens de taak van de vertaler en linkage editor bekijken. De taak van het laadprogramma achten we in het bovenstaande voldoende beschreven.

We spreken in het vervolg alleen over de vertaler en we bedoelen daarmee vertaal- of assembleerprogramma, al naar gelang de brontaal een hogere programmeertaal is of de symbolische machinetaal.

De taak van de vertaler bestaat uit:

- het omzetten van de statements in rijen machine-opdrachten;
- het bijhouden van de adressen van de machine-opdrachten en de variabelen;
- het invullen van de adressen in de machine-opdrachten.

Uit deze taakomschrijving blijkt dat de vertaler een adresseringsprobleem moet oplossen, namelijk het toekennen van adressen. Zoals bekend komen de machine-opdrachten uiteindelijk terecht in het geheugen van de computer en wel tezamen met een aantal andere procesrepresentaties. Het is voor de vertaler natuurlijk niet mogelijk om tijdens het vertaalproces al vast te stellen, waar de opdrachten en variabelen in werkelijkheid terecht zullen komen. De vertaler kan de uiteindelijke adressen niet invullen, hij moet volstaan met een eigen lokale of logische adressering. De vertaler doet alsof ieder programma een eigen privé-geheugen heeft, waarin kan worden geadresseerd vanaf adres 0 of, in een paginageheugen, vanaf paginanummer 0. De vertaler kent dus aan machine-opdrachten en variabelen virtuele adressen toe; tijdens de uitvoering van het programma zorgt de memory manager ervoor, dat deze virtuele adressen worden omgezet in fysieke adressen. Op deze wijze is het adresseringsprobleem van de vertaler op eenvoudige wijze op te lossen.

Er blijft als taak van de vertaler nog de omzetting van de statements in machine-opdrachten. De meeste statements kunnen door de vertaler direct worden omgezet in één of meer machine-opdrachten die het beoogde resultaat van de statements kunnen realiseren. Er is echter een aantal statements die niet rechtstreeks in direct uitvoerbare machine-opdrachten kunnen worden omgezet. Deze statements zijn:
- aanroepen van standaardprocedures, bijvoorbeeld sinus, cosinus;
- aanroepen van eerder ontworpen en apart vertaalde procedures (separate compilation);
- statements die door de vertaler worden vertaald in aanroepen van standaardprocedures zoals input/output statements.

De programmamodulen van de standaardprocedures bevinden zich in de systeembibliotheek. Voordat het gebruikersprogramma (dat we in het vervolg hoofdprogramma zullen noemen) kan worden uitgevoerd, moeten eerst deze standaardprocedures (en apart vertaalde procedures) aan het hoofdprogramma worden toegevoegd.

Procedures kunnen aan het hoofdprogramma worden toegevoegd:

a. voor of tijdens het vertaalproces; de procedures worden dan in onvertaalde vorm in het hoofdprogramma opgenomen;
b. na het vertaalproces; de procedures worden dan in vertaalde vorm in de vertaalde versie (in doeltaal) van het hoofdprogramma opgenomen.

In beide gevallen wordt het samenvoegen van hoofd- en submodulen gerealiseerd door een standaardprogramma: de linkage editor.

De taak van de linkage editor kunnen we samenvatten als: het samenvoegen van een aantal doelmodulen tot een laadmodule (hij verbindt ('linkt') en verzorgt de opmaak).

De invoer voor de linkage editor is een hoofdprogramma (doel-

module). Deze module is door de vertaler gemaakt. Tijdens dit vertalen heeft de vertaler informatie in de module achtergelaten voor de linkage editor:

- op een aantal plaatsen in de code van de doeltaal aanduidingen voor de linkage editor dat een procedure-aanroep moet worden ingevuld, met vermelding van de naam van de procedure;
- een lijst van procedures die in het programma worden aangeroepen.

De activiteiten van de linkage editor kunnen we als volgt omschrijven:

a. Kopieën van de proceduremodulen toevoegen aan het hoofdprogramma; dit gebeurt aan de hand van de lijst van te binden procedures in het doelmodule.
b. De adressen in de proceduremodulen aanpassen; relatieve adressen ten opzichte van het beginadres van de procedure moeten worden omgezet in relatieve adressen ten opzichte van het beginadres van het hoofdprogramma.
c. De gegevens over omvang van programma en toestandsruimte aanpassen; zowel het programma als de toestandsruimte worden in de uiteindelijke versie, de core image code, groter.
d. De doelmodule van het hoofdprogramma doorlopen en op alle aangegeven plaatsen de juiste procedure-aanroepen invullen.

Deze activiteiten hebben uiteindelijk een laadmodule als resultaat; een programma in uitvoerbare code plus informatie over de omvang van het programma in deze code en over de toestandsruimte.

Uit de bovenomschreven activiteiten van de linkage editor blijkt dat ieder programma uiteindelijk beschikt over eigen kopieën van de procedures die in het programma worden aangeroepen. Met andere woorden, wanneer er twee processen worden uitgevoerd waarin dezelfde procedure wordt aangeroepen, zullen er twee kopieën van de procedure in het geheugen aanwezig zijn. Deze procedures zijn, op misschien enkele adressen na, identiek.

Een vraag die men zich kan stellen is, of het mogelijk is dat beide processen op dezelfde kopie van de procedure opereren, zodat kan worden volstaan met één kopie in het geheugen. Dit is mogelijk, maar het stelt eisen aan de adresseringsmethoden binnen de procedure. Een adres in de procedure mag nu namelijk niet meer worden gerelateerd aan het beginadres van een hoofdprogramma, het moet voor alle hoofdprogramma's die van de procedure gebruikmaken, goed zijn. Realisering van de vereiste adressering kan plaatsvinden met behulp van speciale registers waarin, bij aanroep van de procedure, het beginadres van deze procedure wordt geplaatst, gepaard met relatieve adressering ten opzichte van de inhoud van dit register, binnen de procedure.

We gaan hier niet verder op in; we merken op dat een procedure (eventueel ook programma) die in verschillende processen tegelijk kan worden gebruikt, re-entrant (herbetreedbaar) wordt genoemd. Deze methode kan aanzienlijke vermindering van de geheugenbezetting opleveren, denk bijvoorbeeld aan herbetreedbare vertalers.

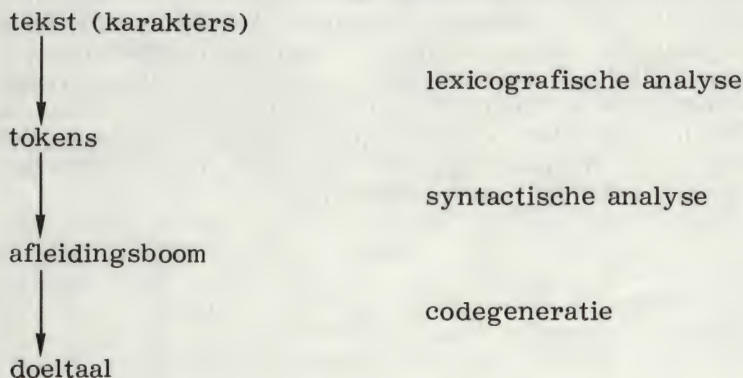
5.9 ENKELE BEGRIPPEN EN TECHNIEKEN MET BETREKKING TOT VERTALERS

Een vertaler is, zoals we hebben gezien, een programma dat een programma in een hogere programmeertaal transformeert in een equivalent programma in machinetaal of een tussenliggende taal. Een zogenaamde 'incremental compiler' vertaalt statement voor statement van een programma zonder het hele programma te kennen. Bij de normale vertalers wordt het programma in zijn geheel geanalyseerd en vertaald. Het resultaat van beide soorten vertalers is een totaal vertaald programma.

De incremental compilers worden gebruikt bij interactieve talen, waarbij de gebruiker zijn programma bijvoorbeeld via een terminal-schrijfmachine aan de vertaler aanbiedt; hij kan dan veranderingen aanbrengen in zijn programma zonder dat dit een nieuwe vertaling van zijn hele programma tot gevolg heeft.

Een *interpretator* (Engels: interpreter) is een programma dat een bronprogramma niet vertaalt, maar direct de betekenis van iedere statement bepaalt en uitvoert. Het resultaat van de interpretator is dus een verwerkt programma. Bij de verwerking van een programma vindt altijd interpretatie plaats.

We kunnen het totale vertaalproces als volgt in beeld brengen:



De tekst kan, zoals hierboven aangegeven, direct geïnterpreteerd worden. Ook is het in tokens gecodeerde programma te interpreteren. Hetzelfde geldt voor de afleidingsboom. Als de doeltaal de machinetaal is, vindt de interpretatie door de hardware plaats.

De machinetaal-opdrachten worden binnen de machine geïnterpreteerd en omgezet in een serie micro-opdrachten, zodanig dat bij iedere machinetaal-opdracht een bepaald microprogramma behoort. Deze microprogramma's kunnen zijn opgeslagen in een read-only geheugen of als schakeling aanwezig zijn.

De verzameling van microprogramma's van de machine noemt men wel de *emulator*. De constructeur van de machine kan de emulator wijzigen en daardoor een andere machine creëren. Men kan hierdoor op machine X machine Y simuleren.

Is de doeltaal een tussentaal, dan zal een software interpreter voor de verwerking zorgen; men spreekt dan van een *simulator*.

Het maken van een vertaler is een moeilijke zaak, omdat het hierbij gaat om een groot programma, dat (in de meeste gevallen) wordt geschreven in machinetaal. Er zijn verschillende mogelijkheden om deze moeilijke taak te verlichten. Allereerst kunnen we denken aan het op een hoger niveau brengen van de taal van de machine, zodat de afstand tussen een voor de mens gemakkelijk te hanteren taal en een taal die door de machine direct kan worden verwerkt, kleiner wordt. Er zijn systemen waarbij dit voor een groot deel is gerealiseerd.

Het is ook mogelijk een tussenliggende taal te ontwerpen met erbij behorende vertalers voor de vertaling van ieder van de hogere talen naar deze taal en een extra vertaler voor de vertaling van deze tussenliggende taal naar machinetaal (deze laatste fase kan ook gerealiseerd worden door een interpreter voor die tussentaal). Als men de tussenliggende taal goed kiest, wordt ieder van de vertalers eenvoudiger en als men de beschikking wil hebben over vertalers voor steeds meer talen in het programmatuursysteem, wordt deze oplossing steeds voordeliger dan de methode van de directe vertaling naar machinetaal voor ieder van de talen. We zouden nu zelfs zover kunnen gaan, dat de tussenliggende taal geheel machine-onafhankelijk is, zodat we maar éénmaal voor iedere taal een vertaler zouden behoeven te maken naar deze tussenliggende taal, en dat de fabrikanten alleen zouden behoeven te zorgen voor één vertaler voor de vertaling van de tussenliggende taal naar hun machinetaal (of voor één interpreter van die tussentaal).

6 EIGENSCHAPPEN VAN PROGRAMMEERTALEN

6.1 INLEIDING

Een programmeertaal dient onder andere voor de communicatie tussen de programmeur en de rekenmachine. De programmeertaal staat tussen de taal uit de toepassingsgebied en de taal van de machine in. Maar waar? De assembleertalen realiseerden een stap, weg van de machinetaal, in de richting van de programmeur. De zogenaamde hogere programmeertalen zijn bedoeld om nog dichter bij de programmeur en zijn problemen te staan. De taal staat echter nog steeds dicht bij de machine, misschien niet wat de notatie betreft, maar wel in de toegestane constructiemogelijkheden. Het leren programmeren wordt dan ook door veel mensen nog vaak gezien als het leren van een programmeertaal. Bij het spreken over een programmeercursus is vaak de eerste vraag: Welke taal wordt er gebruikt? Waarbij gedacht wordt aan talen als BASIC, FORTRAN en Pascal, en waarbij de keuze vaak bepaald wordt door de beschikbare implementatie. De taal moet echter niet afgestemd zijn op de machine, maar de machine moet afgestemd zijn op de taal. De taal in een programmeercursus dient in de eerste plaats om abstracte mechanismen te beschrijven, los van een bestaande machine.

Uiteindelijk zal een programma verwerkt moeten worden op een machine. In de taal die hiervoor gebruikt wordt dringen allerlei machine- en compiler-afhankelijke zaken door. Met het oog op efficiëntie-overwegingen is het misschien ook wel prettig om machine-afhankelijke zaken te kunnen gebruiken. Een oplossing voor de conflicten tussen twee soorten gebruik van een programmeertaal (voor de mens en voor de machine) is wellicht gelegen in het hanteren van twee talen:

- een *ontwerptaal*, die voldoet aan de criteria die aan programmeertalen gesteld kunnen worden met het oog op het gebruik door de mens;
- een *produktietaal*, die voldoet aan criteria die aan programmeertalen gesteld kunnen worden met het oog op de verwerking door de machine.

Door de scheiding kan elk van de talen wellicht beter afgestemd

worden op het bedoelde gebruik. Als verdere voordelen voor deze scheiding zijn nog te noemen:

- de keuze voor een ontwerptaal is onafhankelijk van de beschikbare compilers;
- de ontwerptaal kan uitgebreid worden (aangepast worden aan nieuwe inzichten) zonder dat dit consequenties heeft voor de compilers voor de produktietaal;
- algoritmen kunnen in de ontwerptaal worden geschreven, waardoor in zekere zin overdraagbaarheid verkregen wordt (zo is ALGOL 60 lange tijd gebruikt als dé taal om algoritmen in te publiceren); bovendien kunnen we in de documentatie het algoritme in de ontwerptaal vastleggen en kunnen tegelijkertijd meerdere implementaties in de produktietalen bestaan, waarbij men zijn voordeel kan doen met eigenschappen van ieder van deze produktietalen.

Een groot nadeel van deze opzet is natuurlijk wel dat de programmeur twee verschillende talen moet beheersen en dat er een vertaling gemaakt moet worden van ontwerptaal naar produktietaal. Juist met het oog hierop wordt er gewerkt aan wat genoemd wordt 'wide spectrum languages'. Het gaat hierbij om systemen waar naast produktietalen ook een specificatietaal en een ontwerptaal zijn opgenomen. Van het probleem wordt eerst de specificatie gegeven in de specificatietaal. Daarna wordt het programma getransformeerd. Deze overgangen van het programma in de ene taal naar het overeenkomstige programma in een andere taal kunnen deels volgens formele regels door het systeem worden gedaan en moeten deels door de programmeur zelf worden gedaan.

Welke taal moet nu in het onderwijs gebruikt worden? De beantwoording van deze vraag hangt af van wat men wil onderwijzen. Gaat het om het ontwerpen dan is kennelijk een ontwerptaal de aangewezen taal. Bij het onderwijs geeft het een extra voordeel onderscheid te maken tussen de ontwerptaal en de produktietaal. Men kan de studenten verplichten zelf over de correctheid van hun programma te laten nadenken zonder dat zij door middel van testen al experimenterend tot een correct programma komen.

Een programmeertaal die nogal eens genoemd wordt als het gaat om het onderwijs in programmeren is Pascal. Maar enkele oplossingen die in Pascal zijn gekozen, zijn duidelijk ingegeven door implementatie-overwegingen. Men zou kunnen zeggen dat Pascal een redelijk compromis is tussen een ontwerptaal en een produktietaal. Een groot voordeel van het gebruik van Pascal bij het programmeren ten opzichte van bijvoorbeeld ALGOL 60 is de datastructurering. Het gebruik van Pascal in het onderwijs (als onderwijstaal/ ontwerptaal) biedt één groot voordeel. De meeste nieuwe leerboeken op het gebied van programmeren gebruiken Pascal.

6.2 GEBRUIK VAN DE TAAL

Als we de taal meer willen afstemmen op de programmeur, zouden we kunnen nagaan op welke gebieden de programmeur de meeste hulp nodig heeft en de ontwerpcriteria voor de programmeertaal hieruit afleiden. Gebieden, onderwerpen, die in dit verband genoemd kunnen worden, zijn:

- De probleembeschrijving.
  De programmeur moet de 'technische specificaties' voor het probleem geven. Deze moeten exact zijn. De gebruikte taal moet de mogelijkheid hiertoe bieden. De taal zou echter ook zo moeten zijn dat de opdrachtgever er zijn probleem nog in herkent.
- Het ontwerpproces en het noteren van het programma.
  Het moeilijkste onderdeel van de hele programmering is het komen vanuit de specificaties tot het actuele algoritme. Het ontwerpproces moet ondersteund worden door de gebruikte taal. De taal mag bij het coderen niet een extra moeilijkheid vormen. Een beperkte taal zou ons kunnen afhouden van een fraaie oplossing. Een uitgebreide taal kan keuzeproblemen opleveren.
  De wijze van denken bij het oplossen van (programmerings-) problemen wordt mede bepaald door de talen die we gebruiken bij het ontwerpen en vastleggen van de oplossing.
- De correctheid van het programma.
  Een programma kan bij uitvoering tot veel verschillende processen leiden. De programmeur heeft de verantwoordelijkheid voor de correctheid van al deze processen. Deze verantwoordelijkheid kan hij dragen door zich te concentreren op het gemeenschappelijke van al deze processen: de programmatekst. De programmeur zal dan van elke toegestane constructie de volle betekenis moeten kennen; de semantiek van de taal zal vastgelegd moeten zijn.
- De documentatie.
  Vaak wordt gezegd dat de programmeertaal zodanig moet zijn dat programma's 'zelf-documenterend' zijn. De taal zou moeten toestaan dat commentaar in het programma kan worden opgenomen en dat zinvolle namen gekozen kunnen worden. Dit alles om de leesbaarheid van een programma te bevorderen. Deze eisen worden echter vaak ingegeven door de wens om bij het lezen van een programma zelf 'rekenmachine te spelen': en dan gebeurt dit, en daarna dat, en dan ..... . Belangrijk is het vastleggen van het ontwerpproces en de beslissingen en correctheidsbeschouwingen die hierbij een rol hebben gespeeld. Het kiezen van suggestieve namen en het opnemen van commentaar in de programmatekst kunnen leiden tot schijnzekerheden, doordat meer beloofd wordt dan in werkelijkheid wordt waargemaakt. Natuurlijk kunnen zinvolle namen verhelderend werken, maar zij moeten niet komen in de plaats van bovenbedoelde documentatie.
  Het is ook belangrijk dat de programmatekst de bijbehorende

processen weerspiegelt, dat in een programmatekst de processen en deelprocessen herkenbaar zijn. Dit kan onder andere bereikt worden door een grote vrijheid in lay-out voor programmateksten te bieden (en deze niet op te hangen aan bijvoorbeeld kaartindelingen).

- Het onderhoud.

  Het woord onderhoud heeft bij programmeren een specifieke betekenis. Volgens 'Van Dale' is de betekenis van het woord onderhoud: het in goede staat houden. Dat wil dan wel zeggen dat het voorwerp dat onderhouden wordt, oorspronkelijk in goede staat moet zijn. Bij programma's slaat onderhoud echter op het verbeteren van fouten, en daarnaast op het aanpassen van het programma aan gewijzigde specificaties of het veranderen van het programma om te voldoen aan wijzigingen in het computersysteem. In de praktijk neemt het onderhoud een grote plaats in het geheel van programmeeractiviteiten in. Percentages van 60, 75 of nog meer worden genoemd.

6.3 ONTWERPCRITERIA VOOR EEN PROGRAMMEERTAAL

Er bestaan zeer veel verschillende programmeertalen. Een deel van de diversiteit is te verklaren uit de verschillen in toepassingen en machines. Daarnaast spelen ook verschillen in waardering een rol, die men heeft ten aanzien van eigenschappen of karakteristieken die een taal moet bezitten, bijvoorbeeld met het oog op de correctheid van een in die taal geformuleerd programma. In het verleden, toen nog niet zo bekend was welke eigenschappen een taal moest hebben om de programmeur te ondersteunen bij zijn werk, zijn vaak ontwerpcriteria gehanteerd die niets of nauwelijks iets met de in § 6.2 genoemde punten te maken hadden. Een voorbeeld van een taal waar, gezien de ontwerpcriteria, zeer expliciet met de programmeur rekening is gehouden, is de taal voor het schrijven van operating systems die gebruikt wordt bij het project SUE. Als ontwerpcriteria zijn hier onder andere gehanteerd:

- De voornaamste functie van een programmeertaal is de communicatie tussen mensen en het is essentieel dat deze communicatie duidelijk, gemakkelijk en ondubbelzinnig kan verlopen. Juist als software-systemen groter worden, wordt deze communicatie beslissend.
- Van belang bij het gebruik van hogere programmeertalen voor het construeren van systemen is niet zozeer dat fouten en 'slimmigheidjes' vermeden worden als wel de machtigheid die ze de programmeur verschaffen.

We zullen nu eerst eens kijken naar de criteria die gehanteerd zijn bij het ontwerp van enkele veel gebruikte talen. Deze criteria zijn vaak niet expliciet geformuleerd, maar kunnen tussen de regels door in het definiërend document worden teruggevonden. We kijken naar de oorspronkelijke definities, niet naar latere versies van dezelfde talen.

FORTRAN

"The goal of the FORTRAN project was to enable the programmer to specify a numerical procedure using a concise language like that of mathematics, and to obtain automatically from this specifications an efficient 704 program to carry out the procedure. It was expected that such a system would reduce the coding and debugging task to less than one fifth of what it has been."

Verder wordt gesteld:

- "The language of the system is intended to be capable of expressing virtually any numerical procedure."
- "The statement of a program in FORTRAN language rather than in machine code or assembly program language is intended to result in a considerable reduction in the amount of thinking, bookkeeping, writing and time required."
- "It is considerable easier to teach people untrained in the use of computers how to write programs in FORTRAN than it is to teach them machine language."
- "The structure of FORTRAN statements is such that the translator can detect and indicate many errors which may occur in a FORTRAN language program, the nature of the language makes it possible to write programs with far fewer errors than are to be expected in machine languages programs."

Het valt op dat machine-onafhankelijkheid helemaal niet genoemd wordt (zelfs het tegendeel). Hetgeen natuurlijk niet verwonderlijk is als men bedenkt dat het hier om een produkt van een computerfabrikant gaat.

COBOL

"It was agreed that the language must be open-ended and capable of accepting change and amendment, that it should be problem-oriented and machine-independent, that it should use English or pseudo-English and avoid symbolism as far as possible."

Hier is het opmerkelijk dat men kennelijk geen standaard wilde definiëren. Nu wordt wel gesproken van machine-onafhankelijkheid.

PL/I

- "It is a multi-purpose programming language for use in both commercial and scientific applications."
- "The modularity of PL/I provides different combinations of language facilities for different applications and different levels of complexity. A programmer using a particular combination need not even know about the unused facilities."

- "If a particular combination of symbols has a useful meaning, that meaning is allowed."

Er wordt niets gezegd over gemakkelijk gebruik, over implementatiezaken, over onderwijs, over foutgevoeligheid en machine-onafhankelijkheid.

Pascal

Als ontwerpcriteria voor Pascal zijn gehanteerd dat het een taal moest worden die speciaal geschikt zou zijn voor het onderwijs in het programmeren en dat de taal zodanig moest zijn dat er betrouwbare en efficiënte compilers voor te ontwikkelen zouden zijn.

Ada

We kijken nu eens naar de specificaties voor een nieuwe taal. Door het Ministerie van Defensie van de Verenigde Staten is enige tijd geleden een rapport opgesteld met de technische eisen die aan een nieuwe programmeertaal gesteld worden. Op basis van deze eisen is inmiddels een programmeertaal gedefinieerd: Ada. De algemene eisen zijn:

"

- Generality.
  The language shall provide generality only to the extend necessary to satisfy the requirements.
- Reliability.
  The language should aid the design and development of reliable programs. The language shall be designed to avoid error prone features and to maximize automatic detection of programming errors. The language shall require some redundant, but not duplicative, specifications in programs. Translators shall produce explanatory diagnostic and warning messages, but shall not attempt to correct programming errors.
- Maintainability.
  It should promote ease of program maintenance. It should emphasize program readability over writability, [...] The language should encourage user documentation of programs.
- Efficiency.
  The language design should aid the production of efficient object programs. Constructs that have unexpectedly expensive or inexpensive implementations should be easily recognizable by translators and by users. [...] Execution time support packages of the language shall not be included in object code unless they are called.
- Simplicity.
  The language should not contain unnecessary complexity. It should have a consistent semantic structure that minimizes the number of underlying concepts. [...] The language should have uniform syntactic conventions. [...]

- Implementability.
  The language shall be composed from features that are understood and can be implemented. The semantics of each feature should be sufficiently well specified and understandable. [...]
- Machine Independence.
  The language shall strive for machine independence. [...]
  There shall be a facility for specifying those portions of programs that are dependent on the object machine configuration. [...]
- Formal Definition.
  To the extend that a formal definition assists in achieving the above goals, the language shall be formally defined."

Het rapport van zo'n zestien bladzijden gaat daarna verder en stelt vast welke eigenschappen de afzonderlijke constructies moeten hebben. De bovenstaande eisen zijn niet allemaal gelijksoortig en onder hetzelfde hoofdpunt worden soms zowel eisen genoemd die betrekking hebben op de vertaler als eisen die betrekking hebben op het programmeren.

In de volgende paragrafen komen we terug op de eisen die aan een programmeertaal gesteld kunnen worden.

6.4 EISEN AAN PROGRAMMEERTALEN

Laten we proberen de eisen aan programmeertalen te verdelen in drie categorieën:
a. Eisen met betrekking tot het programmeren;
b. eisen met betrekking tot de compiler, naar de gebruikter toe;
c. eisen met betrekking tot de compiler, naar de machine toe.

Ons interesseren hier vooral de eisen uit categorie a. Voor we hier op ingaan enkele opmerkingen over de andere twee soorten eisen.

b. Compiler/Gebruiker
Van de compiler wordt een goede diagnostiek verwacht. De taal moet dan zodanig zijn dat dit mogelijk is. De foutmeldingen moeten gegeven worden in termen van de taal. De compiler moet fouten niet verbeteren.

c. Compiler/Machine
Onder dit punt vallen onder andere het gemak waarmee de taal geïmplementeerd kan worden, de snelheid van het vertaalproces en de efficiëntie van de gegenereerde code ten aanzien van snelheid en geheugenbeslag.

Nu categorie a. We zullen een aantal eisen noemen, waarvoor geldt dat we zeker niet volledig zijn en waarbij de eisen ook niet geheel onafhankelijk zijn van elkaar.

a.1 *Eenvoudig*
Een programmeertaal moet niet teveel verschillende concepten bevatten. Deze concepten moeten duidelijk en machtig zijn en ze moeten semantisch goed beschreven zijn. Het in een programma samenstellen en combineren van deze concepten moet zonder al te veel beperkingen en uitzonderingen mogelijk zijn (*orthogonaliteit*); dit moet geen onverwachte resultaten opleveren.

Een grote taal maakt het moeilijk de taal 'te doorzien', maakt het voor de compilerbouwer moeilijk een compacte, betrouwbare en efficiënte compiler te maken. Maar bovenal maakt een grote taal het voor de programmeur moeilijk zijn gereedschap te beheersen. Om er voor te zorgen dat er niet experimenteel geprogrammeerd wordt (probeer het maar en kijk wat er gebeurt) moet de programmeur de taal kennen in al zijn finesses. Machtige constructies zijn alleen toelaatbaar als ze op eenvoudige wijze correct zijn te gebruiken. De taal moet het maken van eenvoudige en elegante programma's bevorderen.

Ter verdediging van grote programmeertalen wordt vaak gesteld dat, omdat er allerlei constructiemogelijkheden voor speciale toepassingen in de taal zijn opgenomen, de programmeur zich kan beperken tot een subset van de taal en daardoor toch kan beschikken over een kleinere taal (zie PL/I). Dit is misschien waar zolang de programmeur correcte programma's schrijft binnen de subset. Wijkt hij (niet bedoeld) af van de subset, dan zijn de reacties van het systeem op zijn minst onverwacht.

Eenvoud is echter in de wereld van de programmering niet populair, het is juist de complexiteit die vaak bewondering afdwingt. Taalontwerpers laten juist aan de hand van ingewikkelde voorbeelden zien hoe fraai hun constructies wel zijn. Ook programmeurs maken zich hier schuldig aan. Deze houding ten opzichte van taalconstructies vinden we ook sterk terug bij APL-programmeurs die naar zo kort mogelijke programma's (one-liners) streven.

Onder eenvoud wordt ook wel eens verstaan dat men het programma met een zo klein mogelijk aantal (verschillende) symbolen kan schrijven. Deze eenvoud staat ons niet voor ogen, het gemak bij het lezen en ontwerpen dient voorkeur te hebben boven het gemak bij het schrijven.

a.2 *Consistent*
Verschillen in effect zouden ook door verschillen in notatie tot uitdrukking moeten komen. Er moeten geen onverwachte uitzonderingsgevallen zijn. Gelijke concepten moeten eenzelfde notatie hebben. Het op verschillende wijzen kunnen noteren van hetzelfde concept (dezelfde constructiemogelijkheid) is wellicht bij het schrijven (het kunnen gebruiken van een verkorte schrijfwijze) gemakkelijk, de leesbaarheid komt het zeker niet ten goede. Soms ook zijn bepaalde combinaties wel toegestaan en andere niet (selectie van een component van een record in COBOL). In Pascal wordt de '+' zowel voor de optelling als voor de vereniging gebruikt.

a.3 *Gestructureerd*
De syntactische mogelijkheden van de taal moeten het ontwerpproces ondersteunen en de semantische eenheden op een eenvoudige wijze kunnen weergeven.

Het moet mogelijk zijn deelproblemen op te lossen met behulp van deelprocessen die een eigen lokale toestandsruimte hebben en waarvan de interface met de omgeving zo klein mogelijk is en in de tekst als zodanig is aangegeven. Het effect van procedures moet begrepen kunnen worden los van de omgeving waarin ze geactiveerd worden (afgezien van de parameters). In Pascal bijvoorbeeld is het mogelijk nieuwe typen met een naam te introduceren. In de procedures kunnen we deze typen echter niet op deze wijze invoeren. Het gevolg is dat de procedure niet los van zijn omgeving begrepen kan worden. Als de taal de hier bedoelde eigenschap heeft, komt dit zowel de begrijpelijkheid als de aanpasbaarheid van het programma ten goede.

a.4 *Machine-onafhankelijk*
Hiermee wordt bedoeld dat programma's helemaal begrepen moeten kunnen worden zonder dat er aan de compiler of machine gerefereerd behoeft te worden. Niet de compiler, maar het definiërend document bepaalt de taal. Waar onafhankelijkheid niet mogelijk is (range van waarden), moeten deze eigenschappen in het programma opvraagbaar zijn (kleinste getalwaarde).

Nog steeds is het zo dat de programmeur voor het schrijven van zijn programma's vaak niet alleen de programmeertaal moet kennen, maar ook de machine waarop het programma verwerkt wordt.

a.5 *Betrouwbaar*
Met betrouwbaarheid bedoelen we dat de taal 'uitnodigt' tot het schrijven van correcte programma's. Hierop zijn veel factoren van invloed. We stippen er enkele aan.

- Een programma kan op vele manieren fout zijn. Het programma kan fout zijn omdat de specificaties niet correct zijn of verkeerd zijn begrepen. De programmeur kan fouten maken tegen de syntaxis of semantiek. Al deze fouten zijn natuurlijk niet te voorkomen door het gebruik van een programmeertaal die goede eigenschappen heeft, maar de kans op fouten is daardoor wel kleiner.
- Het begrip betrouwbaarheid van een programma hangt ten nauwste samen met de correctheid van het programma en met de wijze waarop het programma reageert op niet bedoelde invloeden, zoals verkeerde invoer (*robuustheid*). De taal kan hiervoor constructies leveren, bijvoorbeeld door 'exception handling' mogelijk te maken.
- Al eerder zijn enige punten genoemd die samenhangen met betrouwbaarheid, zoals leesbaarheid en gestructureerdheid. Ook de constructies zelf kunnen onbetrouwbaarheid in de hand werken. Weliswaar is niet echt te bewijzen dat bepaalde constructies de

betrouwbaarheid nadelig beïnvloeden, maar in de literatuur kan men genoeg aanwijzingen vinden voor deze veronderstelling (denk aan sprongopdrachten, pointers, globale variabelen, etc.).

- Typering van alle waarden is ook een belangrijk punt. De compiler kan hierdoor een groot aantal fouten op het spoor komen.
- De programmeertaal moet ook zodanig zijn dat op een eenvoudige wijze is na te gaan of een programma correct is. Dit houdt in dat de semantiek goed moet zijn vastgelegd. En tevens dat het mogelijk moet zijn op een gemakkelijke manier over het effect van een programma en zijn onderdelen te spreken.
- Een laatste punt dat genoemd kan worden is de zogenaamde continuïteit. Hiermee wordt bedoeld dat kleine vergissingen niet moeten leiden tot toch (syntactisch) correcte programma's, die een heel ander effect hebben dan oorspronkelijk de bedoeling was. Een klassiek voorbeeld is de FORTRAN-statement:

```
DO 3 I = 1.3
```

  die syntactisch correct is maar waar de punt een komma had moeten zijn. Door deze vergissing krijgt de bedoelde repetitie een heel ander effect.

Tenslotte nog de opmerking dat door managers en gebruikers het belang van de betrouwbaarheid bij het voorschrijven of kiezen van een programmeertaal nog te vaak onderschat wordt. Het hoge percentage programmeringsarbeid dat aan onderhoud wordt besteed zou hier wel eens ten nauwste mee kunnen samenhangen.

Als afsluiting van deze paragraaf nemen we de volgende raadgevingen over die J. Horning heeft gegeven aan taalontwerpers en taalgebruikers:
"Simplicity is a considerable virtue.
When in doubt, leave it out.
Correctness is a compile-time property.
The primary goal pf a programming language is accurate communication to human readers.
Avoid 'power' if it's hard to explain or understand.
If anything can go wrong, it will.
Reliability matters."

7 ELEMENTEN VAN PROGRAMMEERTALEN

7.1 INLEIDING

In dit hoofdstuk zullen een aantal constructiemogelijkheden uit programmeertalen de revue passeren. Aandacht zal worden besteed aan data, besturingsstructuren, subprogramma's en parameteroverdracht. Daarna wordt aandacht besteed aan mogelijkheden voor parallelle programmering. Tenslotte komen een paar aspecten met betrekking tot de correctheid van programma's aan de orde.

De structuur van een *programma* is in het algemeen:
- definitie van typen
- declaratie van de variabelen
- declaratie van de procedures
- het algoritme: de rij statements die het uit te voeren proces beschrijven (d.w.z. de toestandstransformatie realiseren).

7.2 DATA

Typen

Voor veel programmeertalen geldt dat iedere variabele en iedere constante in een programma van een bepaald type moet zijn. Het type bepaalt de waardenverzameling en de operaties. Vaak zijn in de talen een aantal standaardtypen (zoals integer en boolean) opgenomen. Daarnaast komen in de meeste talen structureringsmogelijkheden (zoals array) voor om nieuwe typen te introduceren. Bovendien bestaan in een aantal talen nog andere mogelijkheden om nieuwe typen te introduceren (bijvoorbeeld opsomming, abstracte datatypen).

De typen kunnen worden onderverdeeld in enkelvoudige en samengestelde typen. Bij enkelvoudige typen bestaat de waardenverzameling uit waarden die op het niveau van beschouwing niet uit componenten bestaan, ze vormen altijd één geheel. Alle operaties (afgezien van relatie-operaties) op waarden van deze typen leveren waarden van dit zelfde type op. Bij samengestelde typen bestaat de waardenverzameling uit waarden waarvan componenten geselecteerd kunnen worden.

Een programmeur moet bij het programmeren beslissen of hij de datastructurering van de taal gebruikt of (als het gewenste type

niet in de programmeertaal bestaat) dat hij zijn eigen typen introduceert die dan later geïmplementeerd moeten worden. Aan implementaties zal hier echter geen aandacht worden besteed.

Definitie van constanten

In een aantal talen is het mogelijk een naam te verbinden aan een constante door middel van de definitie van een constante:

const pi = 3.14

Na deze definitie wordt voor elk voorkomen van de naam pi de constante 3.14 gelezen. Uiteraard kan aan de naam van een constante geen andere waarde worden toegekend dan die uit de definitie.

Declaratie van variabelen

In veel programmeertalen moeten alle variabelen gedeclareerd worden. Door de declaratie (typering) kan de programmeur de waardenverzameling en de operaties aanpassen aan de betekenis van de variabele. Daarnaast geeft de declaratie ook aan de vertaler de nodige gegevens. Hierdoor worden mogelijk:

- Efficiënte opslag en access van gegevens.
  De declaratie legt de invariante eigenschappen van de variabelen vast die gelden tijdens de executie van het programma (type elementen, toegestane operaties en dergelijke).
- Een goed geheugenbeheer.
  Het moment van creatie en eventueel ook het moment van vernietiging kunnen door de declaratie zijn vastgelegd. (We komen hier later nog op terug bij de 'blokstructuur'.)
- Statische type-controle (vóór de uitvoering van het programma).
  Dit is mogelijk als ook van iedere waarde eenduidig is vast te stellen wat het type ervan is. Wat de operaties betreft bestaan er dan wel verschillen:
  - operatoren die altijd operanden van een bepaald type eisen en waarbij ook het type van het resultaat hetzelfde is, bijvoorbeeld de 'logische en': $a \wedge b$;
  - operatoren die voor verschillende typen operanden zijn toegestaan; zo is de operator '+' zowel voor het type integer als voor het type real gedefinieerd. Men spreekt in zo'n geval wel van '*overlaoding*'.

  Als statische type-controle mogelijk is wordt de taal *strongly typed* genoemd.

Als variabelen niet gedeclareerd behoeven te worden is de flexibiliteit voor de programmeur groter maar daarmee is alleen dynamische type-controle tijdens de executie van het programma mogelijk. Bovendien is flexibiliteit een gevaarlijke vrijheid voor de programmeur wat de correctheid van het programma betreft.

Sommige programmeertalen waarin expliciete declaraties niet verplicht zijn, kennen wel impliciete declaraties (in FORTRAN kan de naam van een variabele bepalend zijn voor het type).

Operaties
Veel aspecten met betrekking tot data zijn sterk afhankelijk van de operaties op die data. Een type wordt dan ook gedefinieerd door de waardenverzameling plus de operaties.

Voor ieder type zijn als operaties gedefinieerd de waarde toekenning aan een variabele en de test op gelijkheid. Soms echter wordt in een taal onderscheid gemaakt tussen assigneerbare en niet-assigneerbare typen. Bij de laatste soort is een assignment als a := b waarin a en b variabelen zijn, niet toegestaan. (In veel talen is zo'n assignment bijvoorbeeld niet toegestaan bij arrays.)

Voor samengestelde typen zijn nog gedefinieerd de operaties selectie (kiezen van een component-waarde) en constructie (een waarde opbouwen uit zijn componenten).

OPMERKING
Een belangrijke operatie voor iedere datastructuur is: het access (toegang) tot de structuur. Er wordt wel onderscheid gemaakt tussen value-accessing (bijvoorbeeld de waarde van a in a+7) en location-accessing (bijvoorbeeld het adres van a in a := ...); er zijn programmeertalen die in de syntaxis-regels met dit verschil rekening houden, semantisch is er een groot verschil.

7.2.1 Datatypen

In het onderstaande wordt een overzicht gegeven van typen die zoal voorkomen in programmeertalen en van de wijzen waarop nieuwe typen gedefinieerd kunnen worden. Voor een systematische indeling van de structureringsmogelijkheden waarmee samengestelde datatypen gevormd kunnen worden, zou gebruik kunnen worden gemaakt van de volgende criteria:
- Is de structuur homogeen (zijn alle componenten van hetzelfde type)?
- Zijn de elementen van een elementair type of mogen deze zelf weer samengesteld zijn? Mogen de elementen zowel elementair als samengesteld zijn?
- Mogen de waardenverzamelingen van de typen van de samenstellende elementen een willekeurige cardinaliteit hebben?
- Zijn de elementen geordend en hoe is die ordening?
- Is het aantal elementen vast, variabel met een bovengrens of volledig variabel (statische, semi-dynamische en dynamische typen)?
- Hoe worden de elementen onderscheiden (wat is de selectiefunctie): is dat met een constante naam, via een pointer of via een waarde van een bepaald type?

Wij zullen een indeling geven waarbij slechts met de volgende criteria voor de samengestelde typen rekening gehouden wordt: homogeen/heterogeen of vaste lengte/variabele lengte. We nemen aan dat

elementen weer samengesteld mogen zijn. Dit houdt ook in dat een element van een type met een vast aantal elementen zelf wel weer een variabel aantal elementen mag hebben.

a Enkelvoudige typen

a.1 *Aritmetische typen*

Men komt meestal twee aritmetische typen tegen in programmeertalen: integer (waardenverzameling: deelverzameling van de verzameling der gehele getallen) en real (waardenverzameling: deelverzameling van de verzameling der reëele getallen); het laatste type wordt nog wel eens onderverdeeld in twee typen: fixed (vast aantal cijfers achter de komma) en floating (vaste relatieve nauwkeurigheid).

Declaraties van variabelen van deze typen zouden kunnen luiden:

<u>var</u> a: integer;
<u>var</u> c: fixed;
<u>var</u> x: floating;

De waardenverzamelingen van deze typen worden (meestal) bepaald door de wijze waarop waarden worden geïmplementeerd en zijn dus machine-afhankelijk en niet vastgelegd in het definiërend document van de taal. Soms bestaat de mogelijkheid om de grootste en de kleinste waarde op te vragen, bijvoorbeeld door max(integer) en min(integer).

Soms ook bestaat de mogelijkheid om zelf een waardenverzameling te definiëren (mits die ligt binnen de door de implementatie voorgeschreven grenzen). Bijvoorbeeld:

<u>var</u> b: integer <u>range</u> -32768..32767;
<u>var</u> d: fixed <u>delta</u> 0.1 <u>range</u> -100.0..100.0;
<u>var</u> y: float <u>digits</u> 8 <u>range</u> -1E30..1E30;

In sommige talen is het mogelijk de waardenverzameling te vergroten (voor integers wordt dan bijvoorbeeld niet één maar een tweetal machinewoorden gebruikt):

<u>var</u> mm: long integer;

De aritmetische typen zijn geordend. De operaties voor deze typen zijn: optelling, aftrekking, vermenigvuldiging, deling (gehele deling, restbepaling) en machtsverheffing. Daarnaast bestaan er in veel programmeertalen nog een aantal operaties in de vorm van functies (bijvoorbeeld de goniometrische functies). Omdat de typen geordend zijn, zijn relatie-operaties mogelijk (bijvoorbeeld $a < 3$). Bij real-typen moet men oppassen met de ordening. Niet ieder tweetal getallen dat in notatie verschilt zal ook twee verschillende machinegetallen opleveren.

a.2 *Logische typen*

In bijna alle talen komt een 'logisch' type voor met de waarden true en false (of bijvoorbeeld T en F). Meestal ook zijn er een aantal

operaties gedefinieerd, zoals de ontkenning, de 'logische en' en de 'logische of'. Het type is (meestal) niet geordend.

In een aantal talen worden de waarden gerepresenteerd door 1 en 0 en kan vermenging optreden met bijvoorbeeld het type integer.

a.3 *Karakter-typen*

Vrij vaak is de waardenverzameling van het karaktertype de verzameling van ASCII-karakters. Ook andere verzamelingen (bijvoorbeeld EBCDIC) komen voor.

De verzameling van karakters zal veelal geordend zijn: de collating sequence. Bestaat er zo'n ordening dan is een operatie ord(c) mogelijk die van ieder karakter c het rangnummer in de ordening oplevert. Ook de inverse functie chr(i) kan voorkomen, die bij rangnummer i het desbetreffende karakter oplevert.

Na deze zogenaamde standaardtypen (predefined types), typen die niet door de programmeur gedefinieerd behoeven te worden, kijken we nu naar mogelijkheden voor de introductie van nieuwe enkelvoudige typen.

a.4 *Opsommingen (enumeraties)*

Een waardenverzameling kan vastgelegd worden door opsomming van de waarden:

```
type dag = (maandag, dinsdag, woensdag, donderdag, vrijdag,
            zaterdag, zondag)
```

In dit geval zijn er geen andere operaties mogelijk dan die welke gebaseerd zijn op de ordening die gegeven is door de volgorde van opsomming, zoals min(dag) (met als waarde maandag), max(dag), suc(maandag) (met als waarde dinsdag), pred(zondag) (met als waarde zaterdag) en vrijdag < zaterdag (met als waarde true).

a.5 *Subranges (intervallen)*

Van een geordende waardenverzameling kan een deelverzameling geselecteerd worden door een interval te nemen. Deze waardenverzameling en de operaties van het oorspronkelijke type vormen dan een nieuw type. In a.1 hebben we hier eigenlijk al voorbeelden van gezien. Andere voorbeelden zijn:

```
type index = 1..100;
type werkdag = maandag..vrijdag;
```

Subranges worden vaak niet als aparte typen gezien in die zin dat een assignment als

```
i := 210 - 160
```

waarin i van het type index is, correct is, ook al komen in de expressie waarden voor die niet tot de subrange behoren. Als het uiteindelijke resultaat van de expressie maar tot de subrange behoort.

a.6 *Verenigingen*
Als waardenverzameling kan ook de vereniging van de waardenverzamelingen van bekende typen optreden. Bijvoorbeeld:

```
type samen = union of (integer, boolean);
```

Het moet nu wel mogelijk zijn na te gaan of een variabele van het type 'samen' een integer-waarde of een logische waarde heeft, dit in verband met de operaties.

a.7 *Verbijzonderingen*
Om de betekenis van een variabele goed te kunnen benadrukken is het soms mogelijk de waardenverzameling met operaties van een bekend type te introduceren als een nieuw type:

```
type appels = new integer;
type peren = new integer;
var a: appels;
var b: peren;
```

Als de operaties, zoals bijvoorbeeld de optelling, sterk gebonden zijn aan het type, is het nu syntactisch onmogelijk appels en peren bij elkaar op te tellen door middel van a + b.

a.8 *Verzamelingen (sets)*
Een verzameling kan opgevat worden als een enkelvoudig type als de toegestane operaties zijn: de vereniging, de doorsnede, het verschil en andere operaties die weer verzamelingen opleveren. Een verzamelingstype zou gedefinieerd kunnen worden door:

```
type verz = set of T;
```

waarin T een reeds bekend type is.

VOORBEELDEN

```
type kleur = (rood, geel, groen, blauw);
type menging = set of kleur;
type verdieping = 0..10;
type liftstop = set of verdieping;
```

Als s een variabele is van het type liftstop en een waarde heeft, kan het drukken op de knoppen voor de derde, zesde en negende verdieping in de waarde van s tot uitdrukking worden gebracht door: s ∪ {3, 6, 9}. (einde voorbeelden)

a.9 *String-typen*
Rijen van waarden, waarbij de operaties weer nieuwe rijen opleveren, worden strings genoemd:

```
type rij = string of T;
```

Operaties zijn vaak: twee rijen samenvoegen tot een nieuwe rij, in een rij een deelrij vervangen door een (andere) rij, en dergelijke.

In sommige programmeertalen komt het begrip karakterrij voor, waarbij een karakterrij ter lengte 1 van het type karakter is. Dit zou een samengesteld type zijn, omdat dan karakters geselecteerd kunnen worden (als component).

b. Pointer-typen

Pointers worden vaak gebruikt voor de representatie van andere typen. Pointer-waarden zijn eigenlijk adressen. Met deze adressen zelf kan niet of nauwelijks gewerkt worden, maar wel met de geheugenelementen behorende bij deze adressen.

In een aantal programmeertalen kan men een pointer alleen laten wijzen naar waarden van één type:

```
type wijzer = pointer to T;
```

Een variabele van het type wijzer kan dan alleen wijzen naar waarden van het type T.

Voor het bekijken van de operaties op pointers gaan we uit van:

```
var p, q: wijzer;
```

Operaties:

- p := nil
  nil is een speciale waarde van ieder pointertype: de pointer p 'wijst naar niets'.
- new(p)
  Er wordt plaats gereserveerd voor een waarde van het type T en er wordt een pointerwaarde gegenereerd van het type wijzer, die naar deze plaats wijst; deze pointerwaarde wordt aan p toegekend. Aan de waarde waar p naar wijst kan gerefereerd worden door p↑ (is van het type T).
  (De met de operatie new(p) vergelijkbare actie voor variabelen van de andere datatypen vindt impliciet plaats bij declaratie.)
- p↑ := 'waarde van het type T'
  Neem bijvoorbeeld voor het type T het type integer. Dan geldt na:

  ```
  new(p); p↑ := 3
  ```

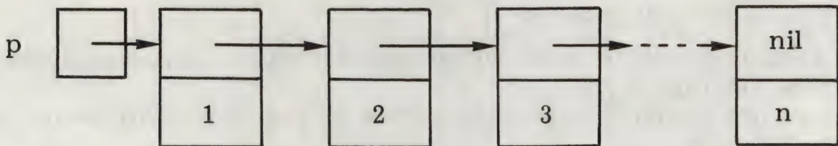
  dat p wijst naar een geheugenelement met de waarde 3.
- dispose(p)
  De ruimte van p↑ wordt vrijgegeven; dit is de tegenhanger van new(p). (Bij variabelen met een statische structuur gebeurt dit automatisch bij beëindiging van de procedure of functie waarin zo'n variabele is gedeclareerd.)
- p := q (p en q van hetzelfde type)
  Als van tevoren geldt p↑ = x en q↑ = y dan geldt na afloop p↑ = y en q↑ = y (p en q wijzen naar dezelfde geheugenplaats).
- p↑ := q↑
  Als vooraf geldt p↑ = x en q↑ = y, dan geldt na afloop p↑ = y en q↑ = y (beide geheugenelementen hebben dezelfde waarde).

VOORBEELD

Stel dat we als type introduceren:

```
type ll = record(volgende: pointer to ll;
                 waarde: integer);
```

Het kenmerk van dit type is dat in de samenstellende records wordt verwezen naar een 'volgend' record; een waarde die op deze wijze is gerealiseerd wordt wel een lineaire lijst genoemd.
Als verwijzing in het laatste element van een lijst wordt de speciale waarde nil genomen. Stel dat er een lijst moet worden opgebouwd van de volgende gedaante:



We introduceren:

```
var p, q: pointer to ll;
```

Als er al een lijst is, kan aan de 'voorkant' een element met waarde k worden toegevoegd door middel van

```
new(q); q↑.waarde := k; q↑.volgende := p;
p := q
```

Voor het algoritme geldt als invariant:
- p↑ is het laatst toegevoegd record
- k is eerstvolgende toe te voegen waarde

Voor het opbouwen van de gehele lijst wordt dan het algoritme:

```
p := nil; k := n;
while k > 0
   do new(q); q↑.waarde := k; q↑.volgende := p;
      {q↑ is nu toegevoegd}
      p := q; {invariant voor p↑ geldt}
      k := k - 1 {invariant voor k geldt}
   od
```

(einde voorbeeld)

c Samengestelde typen

c.1 *Vaste lengte*

c.1.1 HOMOGEEN

'Vaste lengte' slaat op het aantal elementen van de basisstructuur; dat een element zelf weer samengesteld kan zijn en dan eventueel een variabele lengte mag hebben doet hier bij de indeling niet ter zake.

De homogene typen met vaste lengte komen in de meeste programmeertalen in de vorm van *arrays* voor:

<u>type</u> F = <u>array</u> [D] <u>of</u> R;

Een waarde van het type F is een afbeelding (in de vorm van een tabel) van de waardenverzameling van het type D (het indextype) in de waardenverzameling van het type R (het rangetype). Bij elke waarde van het type D hoort een waarde van het type R, aangegeven door a[d] als d van het type D is en a de variabele is:

<u>var</u> a : F;

Voor D zou ieder geordend type mogelijk kunnen zijn (het type real wordt meestal uitgesloten) en voor R ieder willekeurig type. Vaak echter wordt geëist dat R enkelvoudig is en dat de indexverzameling D een rij integer waarden is lopend vanaf 0 of 1. Men spreekt dan ook wel van vectoren:

<u>type</u> v = <u>vector</u> [n] <u>of</u> T;

of van matrices:

<u>type</u> m = <u>matrix</u> [n, m] <u>of</u> T;

Wat de operaties betreft geldt dat we een element van een array kunnen selecteren door bijvoorbeeld a[d] (d wordt de index genoemd). Toekenning van waarden is mogelijk op componentniveau (a[d] := ...) of op variabele-niveau (als in de desbetreffende taal het type assigneerbaar is).

Zoals in het bovenstaande voorbeeld van de matrix kunnen arrays meerdimensionaal zijn (een component wordt geselecteerd op grond van de waarden van meer dan een index). Soms bestaat de beperking tot twee of tot drie dimensies, soms ook zijn willekeurig veel dimensies toegestaan.

c.1.2 HETEROGEEN

Voorbeelden hiervan zijn *records* en structures die in veel programmeertalen voorkomen. Bijvoorbeeld:

<u>type</u> dag = 1..31;
<u>type</u> maand = (januari, februari, maart, april, mei, juni, juli,
augustus, september, oktober, november,
december);
<u>type</u> jaar = 0..2000;
<u>type</u> datum = <u>record</u> (d: dag;
m: maand;
j: jaar);

Een variabele 'dat' van het type datum bestaat uit drie componenten die geselecteerd kunnen worden, bijvoorbeeld door respectievelijk dat.d, dat.m en dat.j. Een waardetoekenning zou kunnen zijn: dat := (7, 10, 1982); ook kan aan een component gerefereerd worden, bijvoorbeeld dat.d := 7 of p := dat.m.

Algemeen geldt:

type r = record (s1 : T1; s2 : T2; ...; sn : Tn)

De waardenverzameling bestaat uit het cartesisch produkt van de waardenverzamelingen van T1, T2, ..., Tn.

c.2 *Variabele lengte*

c.2.1 HOMOGEEN
- Rijen
Omdat de lengte van een rij niet altijd bij voorbaat vaststaat (afhankelijk van invoer, afhankelijk van verwerking) laten programmeertalen rijen van variabele lengte toe. Vaak zijn de rijen wel homogeen. Een voorbeeld hiervan is de *sequence*:

type s = sequence of T;

De waardenverzameling is de verzameling van alle rijen van waarden van het type T.

Operaties zijn bijvoorbeeld het selecteren van de eerste of laatste waarde in de rij en het toevoegen van een waarde aan de rij, aan de 'voorkant' of aan de 'achterkant'.

Er zijn echter een groot aantal verschillende soorten rijen aanwezig in programmeertalen, waarbij het onderscheid wordt gemaakt op grond van de operaties. Aanduidingen die gebruikt worden zijn: stacks, queues, deques, simple lists, strings en files.

- Verzamelingen
Als op een verzameling (*set*) operaties zijn toegestaan als het toevoegen en weglaten van een element, is de verzameling een samengestelde waarde. De operaties zouden kunnen zijn:

any(v): levert een willekeurig element van v als waarde op
delete(x, v): verwijdert uit de verzameling v het element x
(komt overeen met: $v \setminus \{x\}$)
insert(x, v): voegt aan de verzameling v het element met de waarde x toe
(komt overeen met: $v \cup \{x\}$)

c.2.2 HETEROGEEN
Soms zijn rijen toegestaan waarvan de elementen niet van hetzelfde type behoeven te zijn. We zouden deze rijen kunnen aangeven met

type r = sequence of union of (T1, T2, ..., Tn).

7.2.2 Operaties

Een operatie in een programmeertaal heeft als model de wiskundige functie, dat wil zeggen een afbeelding van een domein (invoer) in een range (uitvoer). Dit komt in de denotationele semantiek sterk

tot uitdrukking. Moeilijkheden die optreden om een operatie ook echt als een wiskundige functie te zien, zijn:

- Vaak is in de syntaxis en semantiek van de programmeertaal niet precies vastgesteld wat het domein is (dat wil zeggen dat niet is vastgelegd in welke punten van het domein de functie niet is gedefinieerd).
- Operaties kunnen afhangen van andere zaken dan alleen de expliciete operanden. Dit geldt met name voor procedures (globale variabelen).
- Een operatie kan andere effecten hebben dan alleen in de uitvoer vastgelegd (side-effects in procedures, globale variabelen).
- Een operatie kan zichzelf veranderen, zodat het twee keer toepassen van de operatie tot verschillende resultaten kan leiden (bijvoorbeeld: random number generator).

Iedere programmeertaal kent een aantal primitieve operaties die werken op waarden van standaardtypen. Voor de aritmetische typen zijn dit optellen, aftrekken, vermenigvuldigen, delen (geheel delen, restbepaling) en machtsverheffen. Daarnaast kennen een aantal talen nog extra operaties in de vorm van functies, bijvoorbeeld de goniometrische functies. Deze operaties lijken veel op de overeenkomstige wiskundige functies, maar ze zijn toch niet identiek. Doordat de typen beperkt zijn, zijn ook de operaties beperkt in hun domein en range. Ook de associatieve wet hoeft door deze beperking niet op te gaan. Bovendien is soms de semantiek afwijkend van de wiskundige betekenis; zo geeft in sommige systemen de wortel uit een negatief getal de waarde 0.

Voor geordende typen zijn in veel talen de relatie-operatoren aanwezig ($<$, $\leqq$, $=$, $>$, $\geqq$, $\neq$). Voor het logische type de logische operatoren: conjunctie, disjunctie, ontkenning, implicatie en equivalentie (en soms nog andere, zoals de exclusive-or). Deze operaties zijn ook vaak toepasbaar op bits en bitstrings en doordat in sommige programmeertalen automatische type-transfer operaties aanwezig zijn, zijn hiermee erg onlogische constructies te realiseren.

Ook de type-transfer operaties (conversie van het ene type naar het andere type) kunnen we tot de primitieve operaties rekenen. Soms is de type-transfer expliciet, zoals bijvoorbeeld bij de functie round (die een waarde van het type real afrondt op een waarde van het type integer), vaak echter zijn deze impliciet zoals in ALGOL 60 het geval is bij i1/i2. (In ALGOL 60 is de deling alleen gedefinieerd voor het type real. Komen er integer operanden voor, dan worden deze geconverteerd naar het type real.) Zo'n impliciete typeconversie wordt wel coërcie genoemd.

Ook de assignment kan opgevat worden als een operatie. Over het algemeen eist men een type-gelijkheid, of in ieder geval een type-consistentie, tussen de variabele waaraan de waarde wordt toegekend en de waarde van het rechterlid. Als de typen niet overeenkomen kan men het type van de waarde via een type-transfer

omzetten tot een waarde van het type van de variabele of men kan het type van de variabele aan dat van de waarde aanpassen. Dit laatste kan natuurlijk alleen als de taal geen declaratie van variabelen voorschrijft. (De eerste aanpak wordt bijvoorbeeld gehanteerd in ALGOL 60, de tweede in APL.)

Er bestaan ook operaties voor de creatie en de vernietiging van datastructuren. Deze operaties hangen ten nauwste samen met de geheugenruimte die een waarde inneemt. Bij de creatie moet een verband worden gelegd tussen de gebruikte naam en de plaats waar de waarde wordt vastgelegd (allocatie). De naam kan dan gebruikt worden om acces te realiseren. In plaats van een naam kan ook gebruik worden gemaakt van pointers. Soms gebeurt de creatie impliciet, bijvoorbeeld bij de declaratie van variabelen in ALGOL 60, soms ook expliciet zoals in Pascal voor de plaats waar een pointervariabele naar wijst: new(p). De benodigde ruimte voor variabelen kan in ALGOL 60 dus al bij verwerking van declaraties gereserveerd worden en ook de relatie met de naam kan dan gelegd worden. Voor bijvoorbeeld de pointers in Pascal kan dit pas tijdens de uitvoering van new(p).

Bij het vernietigen moet er goed onderscheid gemaakt worden tussen het ontoegankelijk maken van de structuur en het weer kunnen gebruiken van de geheugenruimte. De momenten waarop deze twee zaken plaatsvinden hoeven niet samen te vallen. Ook hier geldt dat de operatie vaak impliciet gebeurt (bij ALGOL 60 bij blokverlating; we komen daar op terug), soms echter expliciet zoals de dispose(p) in Pascal. Door dispose(p) is het geheugenelement, waar p naar wees, niet meer via p te bereiken; het wordt vrij gegeven (deallocatie). Het gevaar van het vrijgeven van geheugenelementen op deze manier is, dat daardoor eventuele andere toegangspaden naar het geheugenelement wijzen naar iets dat ongedefinieerd is (dangling pointers).

De meeste programmeertalen bieden de mogelijkheid aan de programmeur om zelf nieuwe operaties te definiëren met behulp van functies of functieprocedures.

In sommige talen is het mogelijk nieuwe operatoren te definiëren. Stel bijvoorbeeld:

```
type complex = record (re, im: real);
```

Hiervoor kunnen als operaties gedefinieerd worden:

```
operator "+" (x, y: complex): complex;
    begin return (x.re + y.re, x.im + y.im) end;
operator "*" (x, y: complex): complex;
    begin return (x.re * y.re - x.im * y.im,
                  x.re * y.im + x.im * y.re)
    end
```

Later zouden deze operaties gebruikt kunnen worden in bijvoorbeeld:

```
var a, b, c, d: complex;
a := complex(2.5, 3.5); b := complex(1.4, 1.6).
c := a + b; d := a * b
```

Soms is het ook mogelijk functie-typen te introduceren:

```
type afbeelding = function (real, real) : real;
var p: afbeelding;
```

Aan p kan nu een functie 'toegekend' worden:

```
p(a, b) := a + b
```

Het vastleggen van de volgorde van operaties

Hoe worden nu de operaties en de data gecombineerd tot programma's? Allereerst is de wijze waarop de volgorde van operaties wordt vastgelegd (sequence control) bepalend. Op de tweede plaats gaat het om de overdracht van data tussen de ene en de andere operatie (data control). In de volgende paragraaf komt de data control aan de orde, waarbij de wijze van parameteroverdracht bij procedures een belangrijk onderwerp is.

De regels die de volgorde van de operaties vastleggen (sequence control), kunnen in drie categorieën ingedeeld worden:

- de regels die binnen expressies gelden (*prioriteitsregels*, haakjes);
- de regels voor statements;
- de regels voor de relatie tussen programma en procedure of tussen procedures onderling.

We bekijken nu alleen het eerste punt. Het tweede en derde punt komen in volgende paragrafen aan de orde.

Een *expressie* is een samengestelde operatie die bestaat uit operanden (constanten, variabelen, expressies) gescheiden door operatoren. Gegeven de prioriteiten (van de operatoren) en haakjes en de regel dat bij gelijke prioriteiten de expressie van links naar rechts wordt geëvalueerd (of andersom zoals in APL) kan de waarde van de expressie eenduidig worden vastgesteld. Als de mogelijkheid bestaat om zelf operaties in te voeren moet er natuurlijk ook de mogelijkheid zijn om de prioriteit van deze operaties te definiëren.

Naast de bekende infix-notatie voor expressies komen ook de prefix-notatie (bijvoorbeeld bij functies) en de postfix-notatie voor. Zeker bij de vertaling van expressies. De infix-notatie geeft aanleiding tot een overvloedig gebruik van haakjes. De prefix-notatie (ook wel functionele notatie genoemd) kan leiden tot ingewikkelde constructies.

7.2.3 Abstracte datatypen

Voor de typen in 7.2.1 die door de programmeur gedefinieerd kunnen worden, geldt dat de operaties afgeleid kunnen worden uit een basistype of uit de structuur (array, record, en dergelijke), die bij de definitie is gebruikt. Bij het definiëren van een nieuw type

zou het ook mogelijk moeten zijn de bijbehorende operaties te definiëren.

Stel dat gedefinieerd is:

```
type complex = record (re, im: real);
```

en de operaties daarbij zoals in de vorige paragraaf. Een typedefinitie zou echter moeten zijn het vastleggen van de waardenverzameling plus de operaties, in één definitie dus (type-encapsulation). In een (klein) aantal talen is dit mogelijk. Zo'n definitie kan dan uit twee delen bestaan; het eerste deel geeft aan wat de programmeur van zo'n type moet weten en het tweede deel geeft aan hoe deze zaken zijn geïmplementeerd. We zullen hier, zoals ook elders in dit hoofdstuk, gebruik maken van een fictieve taal (enigszins gebaseerd op Ada). Voor de complexe getallen luidt het eerste deel:

```
module complex-numbers =
   type complex = record (re, im: float);
   operator "+" (x, y: complex): complex;
   operator "*" (x, y: complex): complex;
   operator "×" (x: float; y: complex) : complex
end complex-numbers;
```

Nadat ook het tweede deel is vastgelegd, kan de programmeur bijvoorbeeld gebruiken:

```
var a, b, c, d: complex;
begin a := (2.1, 3.5);
      b := a * a;
      c := a + b;
      d := 4.3 × c
end
```

Daarnaast echter zou de programmeur in staat zijn tot niet zinvol gebruik van complexe getallen als bijvoorbeeld:

```
a.re := 17.1;
b.im := 17.3 * a.re
```

Dit kan doordat de programmeur weet dat het type complex, wat de waardenverzameling betreft, een record-structuur heeft. Dat kan vermeden worden door in het eerste deel van de module-definitie dit gegeven niet op te nemen:

```
module complex-numbers =
   type complex = private;
   operator "+" (x, y: complex): complex;
   operator "*" (x, y: complex): complex;
   operator "×" (x: float; y: complex): complex;
   function makecom(x, y: float): complex
end complex-numbers;
```

We hebben nu een constructie-functie moeten toevoegen, omdat we anders niet in staat zijn een constante van het type complex te noteren.

Het tweede deel (de implementatie) zou nu kunnen luiden:

```
module body complex-numbers =
    type complex = record (re, im: float);
    operator "+" (x, y: complex): complex;
        begin return (x.re + y.re, x.im + y.im)
        end;
    operator "*" (x, y: complex) : complex;
        begin return (x.re * y.re - x.im * y.im),
                      x.re * y.im + x.im * y.re)
        end;
    operator "×"(x: float; y: complex) : complex;
        begin return (x × y.re, x × y.im)
        end;
    function makecom(x, y: float): complex;
        begin return (x, y) end
end complex-numbers;
```

We moeten ons wel realiseren dat we nu niet meer beschikken over de test op gelijkheid. Deze zou in de module gedefinieerd moeten worden.

De toekenning van een constante waarde van het type complex aan een variabele luidt nu:

```
a := makecom(2.1, 3.5)
```

Een tweede voorbeeld realiseert rationale getallen.

```
module rational-numbers =
    type rational = private;
    operator "=" (x, y: rational): boolean;
    operator "+" (x, y: rational): rational;
    operator "*" (x, y: rational): rational;
    operator "/" (x, y: integer): rational
    {dit is de constructiefunctie}
end;
module body rational-numbers =
    type rational = record (numerator : integer;
                            denominator: 1..max(integer));
    procedure same-denominator(var x, y: rational);
        begin ..... end; { geeft x en y dezelfde noemer}
    operator "=" (x, y: rational): boolean;
        begin same-denominator(x, y);
              return (x.numerator = y.numerator)
        end;
```

```
    operator "+" (x, y: rational): rational;
       begin same-denominator(x, y);
             return (x.numerator + y.numerator, x.denominator)
       end;
    operator "*" (x, y: rational): rational;
       begin return (x.numerator * y.numerator,
                     x.denominator * y.denominator)
       end;
    operator "/" (x, y: integer): rational;
       begin return (x, y) end
 end body;
```

De programmeur kan nu bijvoorbeeld gebruiken:

```
 var a, b: rational;
 a := 3/4; b := 6/8;
 if a = b then a := a * b else a := a + b fi
```

Op bovenstaande manier zijn ook stacks te definiëren. Wil men dit doen voor allerlei typen van de elementen van de stack (stack van integers, stack van reals, e.d.), dan zou bij de definitie de mogelijkheid moeten bestaan een type als parameter op te nemen. Het op deze manier gedefinieerde type stack noemt men een *generic type* (of type generator of type-structuur).

7.3 BESTURINGSSTRUCTUREN (sequentiëringsmogelijkheden)

De meest eenvoudige volgorderegel voor statements is dat de een wordt uitgevoerd na de ander, in de volgorde waarin ze in de programmatekst voorkomen (*concatenatie*). De scheiding tussen de achtereenvolgende statements wordt soms expliciet aangegeven in de tekst (in de Inleiding is daarvoor de puntkomma gebruikt), soms geeft de overgang op een nieuwe regel (kaart) de scheiding tussen de statements.

Een tweede mogelijkheid is de *selectie* van één statement uit vele op grond van een selectiecriterium. Vormen hiervan zijn de conditional statement en de *case statement* (het woord statement is hier eigenlijk niet op zijn plaats).

Voorbeelden van de conditional statement (*conditie*) zijn:

```
 if a > b then a := a - b else b := b - a fi;
 if x ≠ 0 then y := 1/x fi
```

De definitie van de conditional statement zou kunnen luiden:

```
<conditional statement> ::=
      if <boolean expression> then <statement list>
                                  [else <statement list>]
      fi
```

Een voorbeeld van de case statement is:

```
case y of
   when -1: x := x + z;
   when  0: x := z;
   when  1: x := x - z;
   others : x := x * z
end case
```

De definitie van de case statement zou kunnen luiden:

```
<case statement> ::=
      case <expression> of
         <case list element>;{<case list element>;}
               others: <statement list> end case
<case list element> ::= <case label list> : <statement list>
<case label list> ::= when <case label> {, <case label>}
```

De semantiek van deze constructie is in de meeste gevallen voor de hand liggend. In sommige talen ontbreekt 'de ontsnappingsclausule' (others...) en het is dan niet altijd duidelijk wat er moet gebeuren als de expressie niet één van de opgegeven waarden heeft. Het is ook niet altijd duidelijk of de case list elements verschillend moeten zijn en als dat niet nodig is of er dan een volgorde is in de keuze.

De derde categorie is de herhalingsmogelijkheid (*repetitie*), die zich in de verschillende programmeertalen op vele manieren manifesteert. In de Inleiding hebben we al de volgende constructie gezien:

```
while <boolean expression> do <statement list> od
```

met als betekenis dat de statement list herhaaldelijk wordt uitgevoerd zolang de boolean expression de waarde true oplevert.

VOORBEELD

```
program division(input, output);
     var y: 1..max(integer);
         x, quotient, remainder: 0..max(integer);
  begin read(x, y);
        remainder := x; quotient := 0;
        while remainder ≥ y
           do remainder := remainder - y;
              quotient := quotient + 1
           od;
        write(x, y, quotient, remainder)
  end
```

(einde voorbeeld)

Bij de while-statement wordt iedere keer voordat de statement list wordt uitgevoerd getest of de boolean expression de waarde true heeft. Een andere mogelijkheid is na iedere uitvoering van de statement list na te gaan of aan een bepaalde voorwaarde wordt voldaan of niet. Een constructie hiervoor is:

```
repeat <statement list> until <boolean expression>
```

De statement list wordt nu ten minste één maal uitgevoerd. De herhaling stopt als de boolean expression de waarde true oplevert. Bijvoorbeeld:

```
x := 1;
repeat x := 0.5 * (x + a/x) until abs(x - a/x) ≦ 1E-6
```

Ook een constructie om een statement list een vast aantal keren te herhalen komt wel voor. De syntaxis voor zo'n soort constructie zou kunnen luiden:

```
<expression> times do <statement list> od
```

Bijvoorbeeld:

```
10 times do read(x); write(x↑2) od
```

In de meeste programmeertalen komt een constructie voor waarmee een statement list een aantal keren wordt herhaald met bij iedere stap een andere waarde voor een variabele, de zogenaamde lopende variabele. We kunnen twee vormen onderscheiden.

Bij de eerste vorm doorloopt de variabele alle waarden in een opgegeven interval. De in Pascal aanwezige constructie van deze vorm kunnen we weergeven als:

```
for <variabele> := <initial expression> to <final expression>
                              do <statement list> od
```

De ondergrens van het interval wordt gegeven door de <initial expression>, de bovengrens door de <final expression>. Bijvoorbeeld:

```
totaal := 0;
for dag := maandag to vrijdag
    do print(dag); read(uren); print(uren);
       totaal := totaal + uren
    od;
print(totaal)
```

Het doorlopen van de waarden van het interval kan 'van laag naar hoog' (zoals in het voorbeeld) of 'van hoog naar laag'.

Bij de tweede vorm is er een grote vrijheid om aan de lopende variabele de waarden te geven waarvoor de statement list wordt uitgevoerd. De in ALGOL 60 aanwezige constructie van deze vorm kunnen we weergeven als:

for <variabele> := <startwaarde> step <stapgrootte>
until <eindwaarde> do <statement list> od

waarin de startwaarde, de stapgrootte en de eindwaarde willekeurige aritmetische expressies mogen zijn.
Bijvoorbeeld:

for x := 1 step i ↑ 2 - i until p * q do ..... od

Met deze vorm van herhaling is voorzichtigheid geboden omdat in de programmeertalen de semantiek nogal kan verschillen. Er zijn talen waar de drie expressies vóór uitvoering van de herhaling geëvalueerd worden en dan verder als constanten optreden en waar ook aan de lopende variabele in de statement list geen waarden mogen worden toegekend; er zijn echter ook talen waar in de statement list de waarden van de expressies beïnvloed kunnen worden door aan er in voorkomende variabelen nieuwe waarden toe te kennen en waar ook nieuwe waarden aan de lopende variabele mogen worden toegekend in de statement list. Voor alle repetities met for geldt, dat er verschillen zijn in de diverse talen (en implementaties) wat de waarde van de lopende variabele na afloop van de repetitie betreft.

In nieuwere talen komt een abstractere vorm van de herhaling voor, in die zin dat de volgorde van waarden voor de lopende variabele niet wordt vastgelegd.
Bijvoorbeeld:

for i ∈ I do s := s + x[i] od

In bijna alle programmeertalen komen een of meer (barokke) mogelijkheden tot een *sprongopdracht* (*goto*-statement) voor. De sprongopdracht verlegt het punt van verwerking naar een andere plaats in de programmatekst zonder op de plaats van vertrek terug te komen. Deze volgordebepaler is nog afkomstig van de mogelijkheid om op machineniveau met de opdrachtteller te manipuleren. Zonder de hele controverse over de sprongopdracht hier te herhalen kan gezegd worden dat met deze opdracht voorzichtigheid is geboden. Vooral omdat de volgorde van de tekst niet meer correspondeert met de volgorde van akties van het proces dat het gevolg is van uitvoering van de tekst. Hierdoor kan de semantiek van het programma erg ingewikkelde worden.

Een beperkte vorm van de sprongopdracht die men ook in nieuwere talen nog wel tegenkomt is de opdracht om 'tussentijds' een herhalingsopdracht te beëindigen door te springen naar de opdracht volgend op deze herhalingsopdracht (*exit*-statement).

In sommige programmeertalen bestaat de mogelijkheid om dynamische fouten (bijvoorbeeld deling door nul) af te handelen (bijvoorbeeld een foutmelding als uitvoer) en daarna de executie van het programma te vervolgen (*exception*-afhandeling). De te ondernemen actie wordt beschreven in een gekenmerkt stukje programma (*exception*-declaratie). Afhankelijk van de taal wordt de executie van het programma ver-

volgd op de plaats waar de exception optrad, of direct ná de module (blok, procedure) waarin de exception gedeclareerd is (zie p.137).

In een aantal talen is een expliciete constructie aanwezig om een statement list aan te geven. Dit gebeurt dan door een rij statements te omsluiten door een openings- en een sluitingshaakje. In ALGOL 60 worden hiervoor de symbolen begin respectievelijk end gebruikt. Deze constructie is in die talen nodig omdat de syntaxis, om redenen van eenduidigheid, op een aantal plaatsen eist dat er één statement staat. Wil men op die plaatsen dan meer statements schrijven, dan worden deze omsloten door het hakenpaar waardoor het één statement wordt: de zogenaamde *compound statement*.

7.4 SUBPROGRAMMA'S, PROCEDURES EN FUNCTIES

7.4.1 Inleiding

We beginnen met een voorbeeld.

```
procedure H(n: integer);
    var i, teller, noemer, h: integer;
    procedure telop(var te, no: integer; g: integer);
        begin te := te * 9 + no;
              no := no * 9
        end;
    begin i := 1; teller := 1; noemer := 1;
          write(1); write(1);
          while i ≠ n
             do i := i + 1;
                telop(teller, noemer, i);
                h := ggd(teller, noemer);
                if h > 1 then teller := teller div h;
                              noemer := noemer div h
                fi;
                write(teller); write(noemer)
             od
    end
```

De procedure drukt bij gegeven n (≧ 1) alle partiële sommen af van

$$\sum_{k=1}^{n} \frac{1}{k} .$$

Een partiële som wordt afgedrukt als een teller en een noemer.

De procedure heeft één (invoer-)parameter. In de body zijn vier lokale variabelen en een lokale procedure gedeclareerd. Verder wordt in de body gebruik gemaakt van een globale (recursieve) functie die gedeclareerd zou kunnen zijn als

```
function ggd(a, b: integer): integer;
  begin if a = b then ggd := a
                 else if a > b then ggd := ggd(a - b, b)
                                else ggd := ggd(a, b - a)
                      fi
        fi
  end
```

Deze functie is *recursief* omdat in de body van de functie de functie zelf weer wordt aangeroepen.

Een *functie* onderscheidt zich van de procedure doordat de samengestelde actie van de body mede tot gevolg heeft dat aan de functienaam een resultaat-waarde wordt toegekend (soort uitvoer-parameter). Deze waarde zal van een bepaald type zijn, en de functie-waarde dus ook. De activering van een functie betekent dat de functiewaarde, afhankelijk van de actuele parameters, berekend moet worden. Het gebruik van de functie bestaat uit het optreden als operand in een expressie.

Het effect van de aanroep van een *procedure* wordt vaak uitgelegd in termen van het kopiëren van de programmatekst op de plaats van aanroep (zoals dat voor een macro geldt).

Een argument dat wel gebruikt wordt voor het afkortingsidee is de winst in geheugenruimte.

Deze kijk op procedures heeft er onder andere toe geleid dat recursieve procedures (directe of indirecte recursie) niet toegestaan of niet nodig geacht werden.

Ook de mogelijkheden en onmogelijkheden van parameteroverdracht zijn vaak ingegeven door deze kopieerregel. Zeer sterk is dit te zien in BASIC waar, afgezien van een speciale sprongopdracht naar de procedure, de procedure gewoon deel uitmaakt van het hoofdprogramma.

Tegenwoordig wordt het proceduremechanisme niet op de eerste plaats gezien als een afkortingsmogelijkheid. Veeleer gaat het om het hiërarchisch opbouwen van een programma, om het realiseren van de verschillende niveaus van abstractie in een groter programma (procedurele abstractie). Als hulp voor de programmeur, niet in de zin van besparing van schrijfwerk, maar als noodzakelijk hulpmiddel bij het ontwerpen, om de complexiteit die bij het programmeren optreedt het hoofd te kunnen bieden.

Bij het werken met procedures moet goed onderscheid gemaakt worden tussen de aanroep (waarbij het alleen gaat om het gewenste effect) en de declaratie (die vastlegt hoe het effect gerealiseerd wordt) van de procedure. Al is het verschil bij een eenvoudige procedure wellicht niet duidelijk en nodig, omdat dan de kopieerregel eventueel toegepast kan worden, bij recursieve aanroepen is het verschil noodzakelijk omdat tegelijkertijd meerdere creaties van dezelfde procedure naast elkaar kunnen bestaan. Iedere activering

creëert een eigen toestandsruimte. Juist vanwege deze eigen toestandsruimte is het, in tegenstelling tot veel andere constructies, zo moeilijk om de verwerking van een recursieve procedure te begrijpen in termen van het uitgevoerde proces.

Bij veel implementaties wordt geen directe ondersteuning in de hardware gegeven en dat leidt dan eventueel tot aanzienlijke overhead. Wat op zijn beurt dan weer tot gevolg heeft dat recursie in een kwaad daglicht komt, terwijl het in een aantal gevallen een zeer bruikbaar programmeergereedschap is.

7.4.2 Het doorgeven, overdragen van gegevens

Een identifier in een programma kan voor veel verschillende zaken staan, bijvoorbeeld: variabele (en dan bepaalt de plaats nog om welke variabele het gaat), constante of procedure. Er moeten dus precieze regels zijn om de betekenis van een identifier te kunnen vaststellen. In veel programmeertalen zijn de regels hiervoor erg ingewikkeld en de uiterste soberheid in het gebruik van de mogelijkheden is dan geboden.

De eerste relatie tussen een identifier en zijn omgeving wordt gelegd bij de uitvoering van de declaratie. Men spreekt wel van de *binding* van de identifier aan zijn omgeving.

De verzameling van actieve, geldige relaties wordt wel de *referentie-omgeving* genoemd. Deze komt dus overeen met de actuele toestandsruimte van het proces. Het stuk programmatekst waarop een binding van toepassing is, wordt de *scope* van de binding genoemd. Een regel die aangeeft welke referentie-omgeving geldig is wordt een scope-regel genoemd. Men onderscheidt dynamische en statische scope-regels. Een dynamische scope-regel bepaalt de betekenis van identifiers in termen van de uitvoering van het programma. Zo wordt in APL de betekenis vastgelegd door het voorkomen van de identifier. Een statische scope-regel definieert de betekenis van identifiers in termen van de structuur van het programma. In Pascal wordt gewerkt met statische scope-regels. De toestandsruimte die ontstaat door de laatste binding wordt de *lokale* referentie-omgeving genoemd. Hetgeen bij deze binding niet verandert in de tot op dat moment heersende referentie-omgeving wordt de *globale* referentie-omgeving genoemd voor het programmadeel waarop de verandering betrekking heeft.

In de declaratie

<u>var</u> i: t

wordt de identifier i gebonden: het is de naam van een variabele van het type t. De identifier t moet al eerder gebonden zijn.

De declaratie van een variabele kan nog een ander effect hebben: er wordt geheugenruimte gereserveerd voor een waarde van het type van de variabele; men noemt dit wel *allocatie*. De *levensduur* (life-

time) van een variabele is de tijdsduur gedurende welke aan de (waarde van de) variabele een geheugenplaats is toegewezen. Vaak kan de toewijzing gebeuren voor de verwerking van het programma (statische allocatie); dit kan bijvoorbeeld voor een integer variabele. Soms kan de toewijzing pas tijdens de verwerking van een programma (dynamische allocatie) plaatsvinden; dit geldt bijvoorbeeld voor een geheugenplaats waar een pointer naar wijst.

Er zullen nu een aantal mogelijkheden voor scope-regels bekeken worden. Op de eerste plaats de *blokstructuur* zoals die in enkele programmeertalen, bijvoorbeeld ALGOL 60 en PL/I, voorkomt. Als een statement kan een zogenaamd blok voorkomen; dit blok bestaat uit een aantal statements voorafgegaan door declaraties. Het geheel wordt in ALGOL 60 omsloten door het hakenpaar begin - end. De referentie-omgeving voor het blok bestaat uit de namen die in het blok gedeclareerd zijn, plus die globale omgeving die bestaat uit namen die niet overeenkomen met namen uit de lokale omgeving. Een variabele is dus globaal ten opzichte van een blok als die variabele in het blok (wel gebruikt wordt maar) niet gedeclareerd is.

```
begin real x, y, z;
      .............
      begin integer v, w, x;
      ............
      end;
      ............
end
```

In het binnenblok (begin integer v, w,x; ..... end) is bijvoorbeeld v een lokale variabele en bijvoorbeeld y een globale variabele. In het binnenblok slaat ieder voorkomen van de naam x op de integer variabele van het binnenblok, de real variabele x is daar onderdrukt maar wordt weer actief bij het verlaten van het binnenblok. Bij het verlaten van het binnenblok worden ook de geheugenplaatsen van de lokale variabelen vrijgegeven.

Er zijn programmeertalen die in dit soort situaties eisen dat van een variabele wordt aangegeven of er aan gerefereerd mag worden in binnenblokken en ook eisen dat in een binnenblok wordt aangegeven aan welke globale variabelen een waarde mag worden toegekend.

Nu worden de scope-regels voor procedures bekeken; die voor functies zijn analoog.

Stel dat we als procedure declareren:

```
procedure P(p: t);
      var h: t;
    begin h := p + x;
          p := h * h
    end
```

Wat stellen de identifiers p, t, h en x voor? p is een parameter, t is

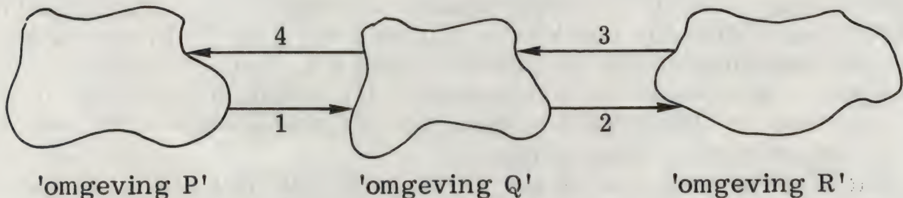
een type, h is een lokale en x een globale variabele. Maar voor welke waarde of type staan ze?

Laten we eens drie procedures P, Q en R beschouwen, waarbij P Q aanroept en Q R aanroept.

```
program var ...
          ..........
          procedure R(...);
              begin ..... end;
          procedure Q(...);
              begin .....; R(...); ..... end;
          procedure P(...);
              begin .....; Q(...); ..... end;
          begin
          .....; P(...); .....
          ..........
          end.
```

We zullen ons concentreren op de omgeving van Q:

1. Welke omgeving wordt er gecreëerd bij de aanroep van Q vanuit P?
2. Wat gebeurt er met de lokale omgeving van Q als vanuit Q R wordt aangeroepen?
3. Wat is de lokale omgeving van Q als er vanuit R wordt teruggekeerd naar Q?
4. Wat gebeurt er met de lokale omgeving van Q als er vanuit Q wordt teruggekeerd naar P?



De vragen 2 en 3 zijn het gemakkelijkst te beantwoorden, omdat de lokale omgeving van Q voor een deel onderdrukt kan worden (zoals bij binnenblokken) bij aanroep van R en de lokale omgeving wordt weer 'actief' bij terugkeer. De geheugentoewijzing blijft onveranderd.

Wat de antwoorden op de vragen 1 en 4 betreft zijn er twee mogelijkheden die dan ook in verschillende programmeertalen worden toegepast.

Bij de eerste methode wordt de lokale omgeving van Q onderdrukt (niet afgebroken) bij terugkeer naar het aanroepende programmadeel (P) en deze omgeving wordt weer actief als Q opnieuw wordt aangeroepen. Dit houdt in dat variabelen, die tijdens een eerdere aanroep een waarde hebben gekregen, deze waarde nog hebben

bij een volgende aanroep. Op deze wijze werkt het proceduremechanisme onder andere in FORTRAN en COBOL. Er is dus een soort symmetrie tussen elkaar aanroepende procedures. Zolang het programmadeel bestaat, bestaan alle lokale omgevingen van de prodecures voor zover deze lokale omgevingen door een aanroep zijn gecreëerd.

Bij de tweede methode wordt de lokale omgeving vernietigd bij terugkeer naar het aanroepende programma en wordt weer een totaal nieuwe omgeving opgezet bij een nieuwe aanroep. Deze methode wordt onder andere toegepast in ALGOL 60, Pascal, LISP en APL. Bij deze methode kun je van een hiërarchie van subprogramma's spreken. Het effect van een procedure is nu (bij ontbreken van globale variabelen) beter te beschrijven omdat dit effect niet kan afhangen van een vorige aanroep. Recursie is bij deze methode beter te realiseren dan bij de eerste methode.

De eerste methode kan gerealiseerd worden door voor iedere procedure tijdens de hele duur van het programma een vaste toestandsruimte te creëren. De tweede methode gebruikt een stack waarop de lokale toestandsruimte wordt gezet bij aanroep, die bij terugkeer weer van de stack wordt verwijderd.

Een speciaal soort procedure met een permanente lokale omgeving is de *coroutine*. In de body van de coroutine komen statements voor die de voortgang van de coroutine stoppen en de executie van de aanroepende eenheid (bijvoorbeeld het eigenlijke programma, of een andere coroutine) laten vervolgen. Als die voortgang overgedragen wordt, wordt vervolgd op de plaats waar de vorige keer afgebroken was. De verschillende programma-eenheden maken dus afwisselend stapsgewijs voortgang; de coroutine is een bruikbaar mechanisme bij discrete simulatie (zie § 9.4.2).

Ook voor de afhandeling van referenties aan een globale omgeving in procedures (geen parameters) bestaan twee mogelijkheden in programmeertalen. Het kan expliciet aangegeven worden door bij de desbetreffende naam op te geven dat het om dezelfde eenheid gaat als die in een andere procedure (of programma). Zo kent PL/I de specificatie EXTERNAL. Ook de COMMON van FORTRAN kan hiervoor gebruikt worden. Als het proceduremechanisme hiërarchisch werkt komt dit expliciet aangeven van globale referenties ook wel voor, maar dan geeft het aanroepende programma dit bijvoorbeeld aan met EXPORT, terwijl het aangeroepen programma de desbetreffende variabelen aangeeft met bijvoorbeeld IMPORT. De tweede afhandelingsmethode voor globale referenties werkt via een impliciet gegeven globale omgeving. Als in een procedure gerefereerd wordt aan een naam die niet in de lokale omgeving voorkomt, wordt verondersteld dat deze naam gevonden kan worden in een globale omgeving. Het vaststellen van de omgeving waarnaar gerefereerd wordt, kan op drie manieren. Stel dat P Q aanroept en dat Q op zijn beurt R aanroept en dat in R een referentie voorkomt aan een niet-lokale x.

```
program var m, x: integer;
        ..........
        procedure R(y);
            begin ..... x ..... end;
        procedure Q(z);
              var a, b: integer;
            begin .....; R(...); ..... end;
        procedure P(t);
              var s, x: integer;
            begin .....; Q(...); ..... end;
        begin ..........
              ..........; P(...); .....
        end.
```

Op de eerste plaats kan de keten van aanroepen in omgekeerde volgorde (te beginnen met Q, dan P, enzovoorts) afgelopen worden en de eerste binding die voor x wordt tegengekomen wordt genomen. (In het voorbeeld de x die in P gedeclareerd is.) Deze werkwijze wordt gevolgd in LISP en APL. Het is een voorbeeld van dynamische binding. De tweede mogelijkheid is dat alleen bindingen uit het hoofdprogramma in de globale omgeving van alle procedures mogen voorkomen. (Dan gaat het in het voorbeeld dus om de x uit het hoofdprogramma.) En tenslotte kan de statische structuur van het programma bepalend zijn. Dat wil zeggen dat niet de plaats van aanroep maar de plaats van declaratie de globale omgeving van een prodecure bepaalt. (ALGOL 60, Pascal en PL/I werken op deze manier.)

In dit geval is in

```
program var x: integer;
        procedure A(y);
            begin ..... x ..... end;
        begin .....
              .....
              var x, z: integer;
              begin .....; A(z); ..... end;
              .....
        end
```

de globale x uit de procedure, die aangeroepen wordt in het binnenblok, de variabele uit het buitenblok waar de procedure gedeclareerd is. Was de procedure in het binnenblok gedeclareerd, dan zou het de x uit het binnenblok geweest zijn. (Let op het verschil met de vorige methode.)

Het expliciet specificeren en de twee laatste methoden, die hierboven genoemd zijn, hebben als eigenschap dat er slechts één globale omgeving is voor een procedure. Het voordeel is de eenvoud van deze methoden: een eenvoudige controle (tijdens vertaling) is mogelijk voor alle referenties, de correspondentie (binding) ligt vast. Het zijn voorbeelden van statische binding (lexical scoping).

De eerste methode van hierboven heeft als nadeel dat pas tijdens

executie de identiteit van de naam (de binding) vastgesteld kan worden en dus ook dynamische type-controle moet plaatsvinden. Voor talen waar dit al noodzakelijk voor is (zoals LISP, SNOBOL en APL) is dat geen probleem, maar voor een taal, waar verder statische typecontrole mogelijk is, is dit een last. Vandaar dat voor een globale referentie voor deze talen de plaats van declaratie of het hoofdprogramma bepalend is.

Welke invloed heeft recursiviteit op de twee methoden van bepaling van de lokale omgeving en de vier methoden voor bepaling van de globale omgeving? Zoals al eerder gezegd is de methode met een 'vaste' lokale omgeving minder geschikt voor recursie. De genoemde methoden voor globale referenties zijn voor recursieve procedures alle vier te gebruiken.

Tot nu toe is alleen gesproken over lokale en globale referenties. Andere namen die een rol spelen in procedures zijn de parameters. Sommige parameters zullen een invoer-functie hebben; de waarde wordt gebruikt tijdens de berekening die bij executie van de procedure wordt uitgevoerd. Andere parameters zullen een uitvoer-functie hebben; aan deze variabelen zullen de resultaat-waarden van de berekening toegekend worden. De parameteroverdracht zorgt voor de relatie tussen formele en actuele parameters, dat wil zeggen voor de data control. Parameteroverdracht geeft een procedure toegang tot niet-lokale elementen via lokale namen. Door het gebruik van parameters (en het afzien van globale referenties) wordt de relatie van de procedure met zijn omgeving precies vastgelegd. Je zou kunnen denken dat het verband tussen actuele en formele parameters niet moeilijk kan zijn: beide komen voor in een lijst en bijvoorbeeld uit de volgorde van de parameters in die lijsten volgt met welke actuele parameter een formele parameter overeenkomt en omgekeerd. Er zijn echter wel een aantal verschillende vormen van parameteroverdracht mogelijk. We zullen de verschillende vormen hier bekijken.

Als voorbeeld nemen we het volgende stukje programma

```
var i : integer;
    x: array[1..3] of integer;
    procedure voorbeeld(<parmech> a, b: integer);
       begin b := b + 1;
             a := 5 * i
       end;
begin for i := 1 to 3 do x[i] := i od;
      i := 2;
      voorbeeld(x[i], i);
      write(i, x[1], x[2], x[3])
end
```

Hierin staat <parmech> voor het soort parametermechanisme dat op a en b (en x[i] en i) van toepassing is.

De eerste vorm die we bekijken is de waarde-overdracht (*value-mechanisme*). Deze vorm kunnen we nog onderverdelen in twee subvormen.

Bij de eerste subvorm wordt bij aanroep de waarde van de actuele parameter bepaald en deze waarde wordt de beginwaarde van de formele parameter. De waarde van de actuele parameter wordt dus gekopieerd in de formele parameter. De formele parameter wordt verder behandeld als een lokale variabele (bij aanroep vindt een allocatie plaats voor de formele parameter). Er wordt dus altijd een waarde overgedragen aan de procedure; omgekeerd kan de procedure de omgeving van aanroep niet beïnvloeden (de actuele parameter behoudt zijn oorspronkelijke waarde). Voor het voorbeeld geldt dat er wordt afgedrukt: 2, 1, 2 en 3.

Bij de tweede subvorm treedt de formele parameter in de procedure op als constante met als waarde de waarde van de actuele parameter die bij aanroep bepaald wordt (nu behoeft er niet gekopieerd te worden). Aan een formele parameter kan nu dus geen waarde worden toegekend in de procedure. In het voorbeeld treedt dus een fout op.

In beide gevallen kan geen waarde overgedragen worden aan het aanroepende programma. Dit value-mechanisme is geschikt voor parameters met een invoer-functie.

Een tweede vorm is de overdracht door referentie (*reference-mechanisme*). Bij de overdracht door referentie wordt een verwijzing naar de plaats van de actuele parameter aan de procedure doorgegeven. Deze plaats wordt bij de aanroep vastgesteld en ook de verwijzing wordt dan vastgelegd. Als de actuele parameter 'geen plaats heeft' (bijvoorbeeld als het een expressie is), wordt een plaats gecreëerd. Vaak echter wordt geëist dat de actuele parameter een variabele is. Iedere referentie aan de formele parameter heeft een acces (lokatie- of waarde-acces) van deze plaats tot gevolg. De waarde van de actuele parameter is dus de beginwaarde van de formele parameter en iedere toekenning aan de formele parameter is in feite een toekenning aan de actuele parameter. Er is transport van waarden van en naar de procedures mogelijk. Het probleem van *aliasing* (zie p.136) kan ontstaan. In het voorbeeld worden afgedrukt: 3, 1, 15 en 3.

Het reference-mechanisme kan zowel voor parameters met een invoer-functie als voor parameters met een uitvoer-functie gebruikt worden; het mechanisme is echter op geen van beide methoden van gebruik toegespitst, zodat zorgvuldigheid geboden is.

Bij naam-overdracht (*name-mechanisme*), de derde vorm, wordt de actuele parameter (de naam, de uitdrukking) gesubstitueerd voor de formele parameter, overal waar deze in de procedure voorkomt. Is de actuele parameter een variabele, dan is de waarde van deze variabele voor de aanroep van de procedure ook de waarde waarmee in de procedure gestart wordt. Iedere toekenning aan de formele

parameter is een toekenning aan de variabele. Is de actuele parameter een constante of een expressie, dan zijn toekenningen aan de parameter niet mogelijk. De substitutie van de actuele voor de formele parameter is een tekstsubstitutie. Is de actuele parameter een expressie, dan wordt (bij de uitvoering van de procedure-body) de waarde van deze expressie steeds opnieuw uitgerekend. Ook het name-mechanisme kan voor beide parameter-functies gebruikt worden. Dat hier erg veel voorzichtigheid geboden is zal blijken uit het volgende.

Een eigenschap van de naam-overdracht die de naam *Jensen's device* heeft gekregen en die door velen als een pluspunt van de naam-overdracht wordt gezien, komt in het volgende voorbeeld tot uitdrukking. Stel dat de functie

```
function som(value m, n: integer; name i, j: integer): integer;
     var s: integer;
   begin s := 0;
         for i := m to n
             do s := s + j od;
         som := s
   end
```

wordt aangeroepen door

```
p := som(1, 100, k, a[k] * b[k])
```

in een omgeving waar a en b arrays zijn met indices van 1 tot en met 100. Het resultaat van de aanroep zal zijn dat p als waarde het produkt heeft van a en b ($p = \sum_{i=1}^{100} a[i] * b[i]$). Dit wordt bereikt doordat de ene actuele parameter, a[k] * b[k], afhankelijk is van de andere actuele parameter, k. Dit effect kan natuurlijk ook bereikt worden door een functie als parameter op te nemen.

Een moeilijkheid die optreedt bij naam-overdracht is dat een door substitutie in de procedure verschenen naam daar al kan voorkomen. Tussen deze namen moet verschil gemaakt worden als de naam in een lokale binding voorkomt, zoals in

```
.....
procedure P( );
       var x: integer;
       begin ..... end;
.....
var x: real;
begin .....; P(x); ..... end;
.....
```

Is het een globale binding, zoals in

```
.....
procedure P ( );
      var a: integer;
      begin ..... x ..... end;
.....
P(x);
.....
```

dan moet nagegaan worden of het dezelfde globale binding is als die van de actuele parameter.

In het voorbeeld van p.119 worden afgedrukt: 3, 1, 2 en 15.

Naast de genoemde drie vormen van overdracht, die het meest gebruikt worden, worden er nog andere vormen onderscheiden: result parameters, value-result parameters en lazy parameters (call by need).

Bij een result parameter (*result-mechanisme*) treedt de formele parameter gedurende de hele procedure op als een lokale variabele (allocatie bij aanroep). Bij terugkeer uit de procedure wordt de waarde die de formele parameter dan heeft toegekend aan de actuele parameter; deze moet dus een variabele zijn. In tegenstelling tot de naam-overdracht is nu dus niet iedere toekenning aan de formele parameter een toekenning aan de actuele parameter. Een constante of expressie als actuele parameter is nu niet mogelijk, bij de naam-overdracht kan dat wel als de formele parameter tenminste alleen in het rechterlid van assignment statements voorkomt. Een probleem ontstaat als twee result parameters in de aanroep hetzelfde zijn: P(a, a). Als de twee ermee corresponderende formele parameters verschillende waarden hebben wat is dan de waarde van a? Een tweede probleem treedt op bij een aanroep als P(a[i], i). Welk array-element krijgt een nieuwe waarde? Voor het voorbeeld zouden result-parameters tot een fout leiden, omdat in de toekenning "b := b + 1" de b in het rechterlid geen waarde heeft.

De value-result parameter (*value-result-mechanisme)* is de combinatie van de eerste subvorm van een value parameter en een result-parameter. De actuele parameter geeft de beginwaarde voor de formele parameter bij aanroep, de formele parameter is verder een lokale variabele variabele en bij terugkeer naar het aanroepende programma wordt de waarde van de formele parameter toegekend aan de actuele parameter. In het voorbeeld wordt afgedrukt: 3, 1, 10 en 3.

Bij de lazy parameter (lazy evaluation) wordt de waarde van de actuele parameter berekend als deze waarde voor het eerst nodig is. Alle andere keren dat deze waarde van de actuele parameter nodig is, wordt deze berekende waarde genomen. Voor het voorbeeld is er geen verschil met de name-parameter (wel als de actuele parameter een expressie zou zijn geweest).

Om een uitspraak te kunnen doen over het effect van een procedure los van zijn omgeving van declaratie en aanroep (globale variabelen

hebben we daartoe al uitgesloten), kan goed gebruik gemaakt worden van value-parameters, result parameters en value-result-parameters. Zijn deze in de taal niet aanwezig, dan kan de programmeur ze met de andere parametermechanismen simuleren.

In veel programmeertalen wordt niet expliciet aangegeven welk parametermechanisme gebruikt wordt en kan het bovendien van de vorm van de actuele parameter afhangen. Voorzichtigheid is dus geboden.

7.5 PARALLELLISME

7.5.1 Inleiding

Er zijn een aantal redenen om mechanismen voor parallelle programmering in talen op te nemen. Enkele redenen zijn:

- Parallelle executie kan de snelheid van uitvoering van een algoritme vergroten. Indien verscheidene processoren tezamen een berekening uitvoeren, zal dat wellicht sneller kunnen gaan dan wanneer slechts één processor aan het werk is. Ook het scheiden van de berekening en de invoer/uitvoer, waarbij de eerste wordt uitgevoerd door een rekenorgaan en de tweede door een I/O-processor, zal in het algemeen de executietijd verminderen.
- Veel real-time toepassingen, zoals vliegtuigreserveringssystemen, database-systemen, procescontrole, etc. vereisen dat een aantal taken zoveel mogelijk parallel uitgevoerd worden. De programmatuur voor deze systemen zal dus op natuurlijke wijze van parallelle mechamismen gebruik kunnen maken.
- Het opsplitsen van een algoritme in een aantal min of meer onafhankelijke delen, die elk de beschrijving zijn van een andere activiteit, zal de duidelijkheid bij ontwerp en implementatie ten goede komen. Een voorbeeld hiervan is het ontwerp van een operating system, waarbij de verschillende taken (virtueel geheugen, invoer/uitvoer, dispatching, etc.) als afzonderlijke (echter wel samenwerkende) processen worden beschreven. Een ander voorbeeld is de simulatie van werkelijke parallelle activiteiten. De beschrijving van deze activiteiten zal dit parallellisme moeten kunnen uitdrukken.

Parallelle programma's beschrijven processen die parallel (simultaan, concurrent) plaatsvinden. Processen vinden parallel plaats als de tijden van executie elkaar overlappen; meer precies: twee processen verlopen parallel als de eerste operatie van het ene proces begint voor de laatste operatie van het andere proces eindigt, en omgekeerd. Daarbij worden geen vooronderstellingen gemaakt over de relatieve snelheden van voortgang. Bij de beschrijving van parallelle processen is het niet relevant of één processor de rekentijd

gaat verdelen over de processen, of dat verschillende processoren elk een proces uitvoeren.

Een groep parallelle processen is te beschouwen als een aantal sequentiële processen die alle apart plaatsvinden. Echter, omdat deze parallelle processen in het algemeen gerealiseerd worden om tezamen een bepaald doel te bereiken, zullen ze moeten samenwerken. Tevens zullen beschikbare middelen door alle processen gedeeld moeten worden, bijvoorbeeld de processen hebben tezamen de beschikking over één regelafdrukker, en die kan kan maar door één proces tegelijkertijd gebruikt worden. Dit betekent dat ze middelen tot hun beschikking moeten hebben om met elkaar te communiceren, en om hun activiteiten op elkaar af te stemmen (synchronisatie, ook te beschouwen als een vorm van communicatie).

De *communicatie* en *synchronisatie* kan gebeuren via gemeenschappelijke variabelen. Het gebruik van gemeenschappelijke variabelen, en ook het gebruik van gemeenschappelijke middelen, zal op basis van exclusiviteit moeten gebeuren, om ongewenste effecten te vermijden. Het gelijktijdig veranderen en inspecteren van de waarde van een gemeenschappelijke variabele zal een onvoorspelbaar eindresultaat opleveren. Het realiseren van die exclusiviteit (*mutual exclusion*), en een goed gebruik ervan, is een van de belangrijkste problemen bij parallelle programmering. De programmeertaal zal mechanismen moeten bevatten om exclusief gebruik van gemeenschappelijke variabelen (in zogenaamde 'kritische secties') te kunnen beschrijven.

De realisatie van exclusiviteit zal tot gevolg hebben dat processen soms moeten wachten tot ze zo'n kritische sectie mogen uitvoeren, omdat een ander proces met de kritische sectie bezig is. Tevens kan het voorkomen dat een proces pas kan voortgaan als een bepaalde synchronisatie of communicatie met een ander proces heeft plaatsgevonden; ook dit kan dus tot wachten leiden. Dit wachten veroorzaakt twee problemen:

- Als de belangstelling om een bepaalde kritische sectie uit te voeren groot is, en dus vaak (steeds) verscheidene processen wachten om de actie uit te voeren, zal de realisatie zodanig moeten zijn dat elk proces op een bepaald moment aan de beurt komt. Kortom, de realisatie van exclusiviteit moet eerlijk zijn, om te vermijden dat (sommige) processen oneindig lang wachten (*individual starvation*).
- Indien processen op elkaar moeten wachten, bijvoorbeeld bij communicatie van boodschappen, moet ervoor gezorgd worden dat dit niet gaat leiden tot een groep processen die cyclisch op elkaar wachten. Bijvoorbeeld: als proces A wacht op een boodschap van proces B, en B wacht op een boodschap van proces C, en C wacht op een boodschap van A, zal geen van de processen A, B, C voortgang kunnen maken; deze drie zullen cyclisch op elkaar blijven wachten, en in deze zogenaamde '*deadlock*' situatie zullen ze alle drie geen voortgang meer kunnen maken. Het vermijden van deze 'deadlocks' is een zeer belangrijk aspect van parallelle programmering.

Voor een parallel programma zal aangetoond moeten worden dat het eindigt. Daar de eindiging afhankelijk is van de eindiging van de afzonderlijke processen, is dat vaak een erg moeilijk aspect.

Ook de correctheid van parallelle programma's (analoog aan de correctheid van sequentiële programma's) zal bewijsbaar moeten zijn. Dit legt eisen op aan de realisatie van 'mutual exclusion', aan de structuur van de gebruikte taal en aan het gebruik van de communicatie- en synchronisatiemechanismen.

7.5.2 Enkele mechanismen

Samenwerkende, parallelle processen zullen voor synchronisatie en communicatie gebruik maken van gemeenschappelijke variabelen: via buffers kunnen boodschappen doorgegeven worden, via waarden van variabelen kan gesynchroniseerd worden. De exclusiviteit bij acces van gemeenschappelijke variabelen wordt gerealiseerd door een gemeenschappelijk mechanisme. Dit gemeenschappelijk mechanisme zal op zichzelf exclusief gebruikt moeten worden om een voorspelbaar effect te realiseren. De programmeertaal zal dus mechanismen moeten bevatten waarbij exclusiviteit gegarandeerd is! Enkele van deze mechanismen worden onderstaand beschreven.

a. Tijdsignalen

De eenvoudigste vorm van synchronisatie is de uitwisseling van tijdsignalen tussen parallelle processen. Het ene proces stopt de voortgang totdat een ander proces een tijdsignaal geeft. Indien een tijdsignaal gegeven wordt als geen proces daarop wacht, heeft dit geen enkel effect. Het volgende programma illustreert de uitwisseling van een tijdsignaal tussen twee processen 'zender' en 'ontvanger' door middel van operaties op een gemeenschappelijke variabele 'e' van het type event. Het geven van een tijdsignaal komt overeen met de operatie 'cause(e)' en het wachten met de operatie 'wait(e)'.

```
zender: .....;                      ontvanger: .....;
        'produceer data';                      wait(e);
        'zet data in buffer';                  'haal data uit buffer';
        cause(e);                              'verwerk data';
        .....                                  .....
```

Voor de operaties 'cause' en 'wait' moet gelden dat ze exclusief de variabele 'e' accesseren.

Het effect van 'cause' en 'wait' operaties hangt af van de volgorde waarin ze uitgevoerd worden. In het voorbeeld geldt dat, indien de ontvanger eerder de operatie 'wait(e)' uitvoert dan de zender de operatie 'cause(e)' de synchronisatie correct verloopt; maar als de zender de operatie 'cause(e)' uitvoert alvorens de ontvanger de operatie 'wait(e)' uitvoert, zal de ontvanger oneindig lang blijven wachten.

Bij dit eenvoudig synchronisatiemechanisme hangt de werking dus af van de relatieve snelheden van de processen; dat maakt het gebruik van tijdsignalen ongeschikt als algemeen mechanisme ter realisatie van communicatie en synchronisatie.

b. Seinpalen

Een beter mechanisme is het gebruik van seinpalen. Seinpalen onderscheiden zich van tijdsignalen doordat het aantal 'nog niet verwerkte' signalen geregistreerd wordt, zodat een 'wait' operatie gerust na zo'n 'cause' operatie mag komen.

Een seinpaal is een bijzondere variabele van het type integer die geen negatieve waarden kan aannemen. Op een seinpaal s zijn twee operaties, P en V, gedefinieerd:

```
P(s): if s = 0 then 'wacht tot een operatie V(s) dit proces wekt' fi;
      s := s - 1
V(s): s := s + 1; 'indien een of meer processen wachten (door uit-
      voering van P(s)) tot s > 0: wek een van hen'
```

P- en V-operaties op eenzelfde seinpaal sluiten elkaar in tijd uit. Voor de keuze van één van de wachtende processen bij V(s) geldt dat deze keuze eerlijk is.

Met behulp van een seinpaal s, die initieel de waarde 0 heeft, kunnen een zenderproces en een ontvangerproces gesynchroniseerd worden:

```
zender: .....;                    ontvanger: .....;
        'produceer data';                    P(s);
        'zet data in buffer';                'haal data uit buffer';
        V(s);                                'verwerk data';
        .....                                .....
```

De commutatieve operaties P en V zorgen ervoor dat relatieve snelheden van processen niet het effect beïnvloeden.

VOORBEELD

Een aantal cyclische processen (P_0, P_1,...,P_n) moeten alle steeds een kritische sectie uitvoeren. De exclusiviteit hierbij kan verzorgd worden door een seinpaal 'mutex', die initieel de waarde 1 heeft:

```
Pi: while true do P(mutex);
                  'kritische sectie';
                  V(mutex);
                  .....
               od
```

VOORBEELD

Een cyclisch proces, de producent, produceert elke slag van de repetitie een hoeveelheid informatie: een boodschap. Een ander cyclisch proces, de consument, zal in elke slag van de repetitie één zo'n boodschap verwerken. Voor de communicatie maken deze beide

processen gebruik van een gemeenschappelijke buffer die N boodschappen kan bevatten. De producent kan dus maximaal N slagen op de consument 'vooruitlopen'. Voor deze synchronisatie en communicatie worden gedeclareerd (betekenis is vastgeled in de naam):

```
var aantalboodschappeninbuffer,
    aantallegeplaatseninbuffer,
    bufferacces: seinpaal;
aantalboodschappeninbuffer := 0;
aantallegeplaatseninbuffer := N;
bufferacces := 1;
```

waarna producent en consument als volgt zijn te beschrijven:

```
producent: while true do 'produceer volgende boodschap';
                         P(aantallegeplaatseninbuffer);
                         P(bufferacces);
                         'voeg boodschap aan buffer toe';
                         V(bufferacces);
                         V(aantalboodschappeninbuffer)
           od
consument: while true do P(aantalboodschappeninbuffer);
                         P(bufferacces);
                         'haal volgende boodschap uit buffer';
                         V(bufferacces);
                         V(aantallegeplaatseninbuffer);
                         'verwerk boodschap'
           od
```

c. Monitors

Seinpalen en de P- en V-operaties zijn erg primitief. Omdat de operaties P en V bij elkaar horen, maar los van elkaar als statement voorkomen (soms zelfs in programma's voor verschillende processen) is de kans op fouten groot. Een ander bezwaar is dat de gemeenschappelijke seinpalen noch tot een van de processen noch tot de feitelijke gemeenschappelijke middelen behoren; het zou prettig zijn als alle gemeenschappelijke zaken bij elkaar behoren.

Monitors komen aan deze bezwaren tegemoet. Het idee is gebaseerd op het class-concept van SIMULA 67.

Een monitor is een blok dat bestaat uit een collectie data (de gemeenschappelijke variabelen), een collectie procedures die deze data manipuleren (de relevante operaties), en een initialisatie van de data. De data kunnen alleen door gebruik van de monitorprocedures geaccesseerd worden. De exclusiviteit bij acces van de data wordt gerealiseerd doordat, indien een van de parallelle processen zo'n monitorprocedure uitvoert, andere aanroepen van procedures van deze monitor opgehouden worden totdat deze geëindigd is; daarna wordt één van de aanroepen gehonoreerd, en de andere blijven wachten, etc.

Indien verscheidene monitoren van hetzelfde type nodig zijn, kan een monitor class (vergelijkbaar met een type) gedefinieerd worden en daarna kunnen variabelen van dat type gedeclareerd worden. Bij declaratie wordt de initialisatie direct uitgevoerd. Het acces tot een gemeenschappelijk middel moet soms opgehouden worden (bijvoorbeeld als een buffer vol is). Omdat alle statusinformatie tot de monitordata behoort, moet er binnen de monitorprocedures een mechanisme zijn om de voortgang tijdelijk te stoppen. Daarvoor bestaat de mogelijkheid om binnen monitors variabelen van het type condition te declareren. Elke variabele van dat type staat voor een reden waarom een proces zou moeten wachten. De operatie 'wait' op een condition variabele c, wait(c), stopt de voortgang van het proces dat deze procedure aanriep. Een operatie signal(c) laat precies één van de processen die wait(c) uitvoeren doorgaan. In verband met de exclusiviteit zijn hierbij twee zaken zeer belangrijk:
- De wait-operatie geeft de monitor vrij, dat wil zeggen een ander proces kan een monitorprocedure gaan uitvoeren.
- De operatie signal(c) betekent het einde van de procedure-aanroep, en heeft tevens tot gevolg dat een van de op condition c wachtende processen hervat wordt. Indien er geen wachtende processen zijn, heeft de signaloperatie alleen tot effect dat de procedure beëindigd wordt.

VOORBEELD
Een type seinpaal, geschikt voor het realiseren van mutual exclusion bij kritische secties, kan beschreven worden door middel van een monitor class declaratie:

```
monitor class seinpaal;
begin var waarde: integer;
      var positief: condition;
      procedure P;
      begin if waarde = 0 then wait(positief) fi;
            waarde := waarde - 1
      end;
      procedure V;
      begin waarde := waarde + 1;
            signal(positief)
      end;
      waarde := 1 {initialisatie}
end seinpaal;
```
(einde voorbeeld)

Declaraties van variabelen van monitor classes geschieden door de naam van de class als type te vermelden.
Bijvoorbeeld:

```
var mutex: seinpaal;
```

Nadat een monitor gedeclareerd is (direct, of met behulp van een

declaratie als variabele van een class) kunnen de procedures aangeroepen worden door middel van de monitornaam en de procedurenaam (eventueel gevolgd door actuele parameters) gescheiden door een punt. Bijvoorbeeld:

```
mutex.P;
mutex.V;
```

Een monitor heeft geen eigen activiteit na de initialisatie. De monitor stelt slechts procedures ter beschikking van de processen en voorziet daarbij in de exclusiviteit van operaties op de data.

VOORBEELD
producent en consument die communiceren via een buffer van N plaatsen:

```
monitor class buffer;
begin var B: array [0...N - 1] of boodschap;
      var eerstelegeplaats: 0...N - 1;
      var aantalboodschappen: 0...N;
      var nietleeg, nietvol: condition;
      procedure voegtoe(m: boodschap);
      begin if aantalboodschappen = N
            then wait(nietvol) fi;
            {0 ≤  aantalboodschappen < N}
            B [eerstelegeplaats] := m;
            eerstelegeplaats := (eerstelegeplaats + 1) mod N;
            aantalboodschappen := aantalboodschappen + 1;
            signal(nietleeg)
      end;
      procedure haaluit(var: m: boodschap);
      begin if aantalboodschappen = 0
            then wait(nietleeg) fi;
            {0 < aantalboodschappen ≤  N}
            m := B [(eerstelegeplaats - aantalboodschappen) mod N]
            signal(nietvol)
      end;
      aantalboodschappen := 0;
      eerstelegeplaats := 0 {initialisatie}
end buffer;
```

Na declaratie van de variabele buf:

```
var buf: buffer;
```

kunnen producent en consument communiceren:

```
producent: while true do  x := 'volgende boodschap';
                          buf.voegtoe(x)
                       od;
```

```
consument: while true do buf.haaluit(y);
                         'verwerk y'
                         od
```
(einde voorbeeld)

Alle zaken die met een bepaalde communicatie of synchronisatie te maken hebben kunnen dus gegroepeerd worden binnen een monitor. Dat maakt monitors erg eenvoudig te gebruiken, en een erg geschikt middel om van irrelevante details te abstraheren.

7.5.3 Enkele talen

De toenemende belangstelling voor parallelle programmering heeft tot gevolg dat enkele nieuwere talen parallelle constructies bevatten. Een tweetal van zulke talen wordt hier besproken.

a. Concurrent Pascal
Concurrent Pascal werkt met variabelen van het type monitor en het type process. Beide moeten als zodanig gedeclareerd worden.

De processen hebben als parameter de monitors waartoe ze toegang hebben, zodat dynamisch monitornamen kunnen worden doorgegeven. Zo wordt vastgelegd welke monitors voor een proces toegankelijk zijn. Initialisatie van monitors en start van executie van processen gebeurt met de init-statement: alle genoemde processen en monitors worden daarna parallel uitgevoerd (voor monitors alleen de initialisatie).

In monitors kunnen twee soorten procedures voorkomen: procedures die alleen binnen de monitor aangeroepen kunnen worden (procedure) en procedures die door processen aangeroepen kunnen worden (procedure entry).

De synchronisatie binnen monitors gebeurt hier met variabelen van het type queue. De wachtoperatie op een queue q is delay(q) en de voortzettingsoperatie continue(q). Tevens bestaat de operatie empty(q) om te inspecteren of een proces op deze conditie wacht. Ten hoogste één proces kan in een queue staan.

VOORBEELD
Een invoerproces (dat kaartbeelden leest van de kaartlezer) en een uitvoerproces (dat kaartbeelden afdrukt op de regeldrukker), die voor de communicatie van kaartbeelden gebruik maken van een buffer van één plaats.

```
program
type card = array [1..80] of char;
     buffer = monitor
              var c: card;
                  vol: boolean;
                  wachtendeinvoer, wachtendeuitvoer: queue;
              procedure entry put(cp: card);
              begin if vol then delay(wachtendeinvoer);
                    c := cp; vol := true;
                    continue(wachtendeuitvoer)
              end;
              procedure entry get(var cc: card);
              begin if ¬vol then delay(wachtendeuitvoer);
                    cc := c; vol := false;
                    continue(wachtendeinvoer)
              end;
              begin vol := false
              end buffer;
     invoer = process (bb: buffer);
              var c1: card;
              begin cycle readcard(c1);
                          bb.put(c1)
                    end
              end;
     uitvoer = process (bb: buffer);
               var c2: card;
               begin cycle bb.get(c2);
                           printline(c2)
                     end
              end;
var b: buffer;
    ip: invoer;
    op: uitvoer;
begin init b, ip(b), op(b) end.
```

b. Ada

In de taal Ada worden parallelle processen beschreven als *tasks*. Tasks bestaan uit een specificatiegedeelte en uit een body. Het specificatiegedeelte beschrijft de operaties ('entries') die voor andere tasks toegankelijk zijn en de body beschrijft de uitvoering van de tasks. Door middel van de 'entries' kunnen tasks dus communiceren. De body bevat variabelen die voor andere tasks niet toegankelijk zijn en statements die uitgevoerd worden als de task geïnitieerd is. Een specificatiegedeelte komt voor in het declaratiegedeelte van een 'program unit' (bijvoorbeeld een subprogramma, een blok, een package, een task) die de ouder genoemd wordt. De body staat in de body van de ouder, of komt voor als 'separate compilation unit'. Zodra de ouder geactiveerd is, worden alle tasks die in het decla-

ratiegedeelte voorkomen, direct geïnitieerd. Dit heeft tot gevolg dat al deze tasks parallel met de ouder worden uitgevoerd. De entries van een task kunnen door andere tasks worden aangeroepen, net zoals een procedure-aanroep gaat. Echter, in verband met synchronisatie geldt dat een entry alleen uitgevoerd kan worden als in de aangeroepen task een accept statement voorkomt. Het uitvoeren van een entry, als gevolg van het samenvallen van een aanroep en een accept, wordt een rendezvous van de beide tasks genoemd; de statements na accept, tussen do en end worden dan uitgevoerd.

VOORBEELD
Een producent communiceert via een één-plaatsbuffer met een consument. De specificatie van de buffer is als volgt:

```
task buffer is
   entry zend(in: in boodschap);
   entry ontvang(uit: out boodschap);
end;
```

En de body:

```
task body buffer is
   b: boodschap;
   begin
      loop
         accept zend(in: in boodschap)
                do b := in end;
         accept ontvang(uit: out boodschap)
                do uit := b end;
      end loop
   end buffer;
```

Een programma dat de producent, consument en buffer definieert en initieert zal er als volgt uit kunnen zien:

```
procedure VB1 is
      'specificatie van producent P';
      'specificatie van consument C';
      'specificatie van buffer';
begin
     .....
     'body van VB1'
     .....
end;
```

Producent en consument kunnen de entries zend en ontvang met actuele parameters aanroepen.
(einde voorbeeld)

Aanroepen van entries, mits niet direct gehonoreerd, worden in een queue geplaatst, en in volgorde van aankomst verwerkt zodra accept statements dit toelaten. Per entry is zo'n queue aanwezig.

De uitvoering van een entry behelst de reeks statements na accept tussen do en end; dit wordt een kritische sectie genoemd. Per task wordt ten hoogste één kritische sectie uitgevoerd. De aanroepende task wordt bij uitvoering van een kritische sectie stopgezet en na uitvoering van de kritische sectie weer voortgezet. Vandaar een correcte synchronisatie en communicatie.

Vaak zal het niet te voorspellen zijn in welke volgorde de verscheidene entry-aanroepen zullen voorkomen. Meer vrijheid bij de volgorde van uitvoering van entry-aanroepen zal aantrekkelijk zijn en daarom bestaat de select statement om een keuze tussen accept statements te laten afhangen van condities en aanroepen.
Bijvoorbeeld:

```
select when ¬vol ⇒ accept zend(...) do...end;...
    or when vol ⇒ accept ontvang(...) do...end;...
   end select;
```

Bij de uitvoering van de select statement worden eerst de condities geëvalueerd en dan ontstaat de verzameling toegestane alternatieven. Indien er meer dan nul toegestane alternatieven zijn wordt er op willekeurige wijze één gekozen voor de voortgang. Indien er geen toegestane alternatieven zijn, wordt dit beschouwd als een fout.

Indien verscheidene processen van dezelfde aard zijn kan door middel van één specificatie en één body de beschrijving van al deze processen gegeven worden.
Bijvoorbeeld:

```
task type producent is
     entry x(...);
     .....
end;
task body producent is;
.....
end producent;
```

Tasks van een gedefinieerd type worden geïnitieerd door een declaratie, bijvoorbeeld: p: producent. De entries van zo'n task kunnen aangeroepen worden met de dot notatie, bijvoorbeeld: p.x. Tasks eindigen als de uitvoering van de body geëindigd is en als alle lokaal geïnitieerde tasks ook geëindigd zijn.

Bovenstaande beschrijving van parallelle aspecten van Ada is niet compleet, maar probeert een indruk te geven van de mogelijkheden. Tot slot een

VOORBEELD

M producenten communiceren via een buffer van T plaatsen met N consumenten. Het programma is als volgt:

```
procedure VB2 is
        task type producent is
               .....
        end;
        task type consument is
               .....
        end;
        task buffer is
           entry zend(in: in boodschap);
           entry ontvang(uit: out boodschap);
        end;
        p = array (1..M) of producent;
        c = array (1..N) of consument;
        begin 'body van VB2' .....
        end;
```

De body van de buffer is nu:

```
task body buffer is
grootte: constant integer := T;
b: array (1..grootte) of boodschap;
volgendein, volgendeuit: integer range 1..T := 1;
aantal: integer range 0..T := 0;
begin
    loop
       select
          when aantal < T ⇒
               accept zend(in: in boodschap) do
                      b(volgendein) := in;
                      end;
                      volgendein := (volgendein mod T) + 1;
                      aantal := aantal + 1
          or
            when aantal > 0 ⇒
                 accept ontvang(uit: out boodschap) do
                     uit := b(volgendeuit);
                 end;
                 volgendeuit := (volgendeuit mod T) + 1;
                 aantal := aantal - 1
      end select;
   end loop;
end buffer;
```

7.6 CONSTRUCTIES DIE DE CORRECTHEID VAN EEN PROGRAMMA IN POSITIEVE ZIN (DIENEN TE) BEÏNVLOEDEN

In de loop der tijd is van een aantal in programmeertalen aanwezige constructies gezegd en geschreven dat zij eigenlijk niet gebruikt zouden moeten worden. Het bekendste voorbeeld is de sprongopdracht waarover al sinds 1968 wordt gedebatteerd. Andere kandidaten zijn de pointer en de globale variabele, zelfs de assignment statement staat ter discussie. Bij al deze onderwerpen gaat het er om de vrijheid van de programmeur te verkleinen om de kans op correcte programma's te vergroten.

Veel aandacht wordt heden ten dage geschonken aan de typering van de waarden in een programma. Talen als Pascal en Ada eisen dat van iedere waarde, of die nu wordt voorgesteld door een constante, variabele of expressie, in het programma is vast te stellen wat het type is (*strong typing*). Dit geeft de mogelijkheid tot zogenaamde statische typecontrole, waardoor veel fouten vroegtijdig kunnen worden opgespoord.

Toch zitten er nog wel wat addertjes onder het gras. Zo hoeft van een functie die als parameter optreedt bij een procedure of een functie in Pascal alleen het type van het resultaat vermeld te worden. Daardoor is pas bij aanroep na te gaan of de actuele (functie) parameter wel het juiste aantal parameters heeft.

VOORBEELD

```
function trap(x, y: real; n: integer; function f: real): real;
     var i: integer;
         sum, z, width: real;
   begin width := (y-x)/n;
         i := 1; sum := 0;
          while i < n do
             begin z := x + i * width;
                   sum := sum + f(z);
                   i := i + 1
             end;
          trap := width * (sum + (f(x) + f(y))/2)
   end
```

Als deze functie wordt aangeroepen met trap(0, 1, 100, sin) zal, door gebruik van de trapeziumregel, de benaderde integraal berekend worden voor de sinus over het interval [0, 1]. Als we als actuele parameter een functie meegeven die zelf twee argumenten heeft (bijvoorbeeld: ggd(a,b)), zal bij verwerking een fout optreden.
(einde voorbeeld)

Kennelijk moeten we òf de functies beter typeren, bijvoorbeeld als f(real): real, òf functies (en procedures) moeten verboden worden als parameter. Deze laatste oplossing is bijvoorbeeld gekozen in de programmeertaal EUCLID.

In een vorige paragraaf is gesproken over de referentie-omgeving voor globale variabelen. De globale variabelen blijven een probleem en in sommige talen worden dan ook regels gegeven voor het gebruik. Zo kan, op eenzelfde manier als bij parameters van procedures, van globale variabelen vastgelegd worden of ze in een binnenblok of elders alleen kunnen optreden als constanten of dat er ook waarden aan mogen worden toegekend (we hebben dit al eerder genoemd). In onderstaand voorbeeld wordt met import en export aangegeven dat het om globale variabelen gaat en wat hun functie is.

VOORBEELD

```
export var z; import x, y;
var a, b, c: integer;
begin b := x; c := y;
      a := 0;
      while b ≠ 0
         do while b mod 2 = 0
               do b := b div 2; c := 2 * c od;
            b := b - 1; a := a + c
         od;
      z := c
end
```

(einde voorbeeld)

Problemen treden ook op als twee namen verwijzen naar een zelfde waarde. In Pascal treedt dit probleem bijvoorbeeld op als twee pointers p en q dezelfde waarde hebben. Als p↑ nu van waarde verandert, dan verandert ook q↑. Een zeer bekend voorbeeld van dit verschijnsel, dat *aliasing* wordt genoemd, treedt ook op als we met verschillende formele parameters dezelfde actuele parameters laten corresponderen.

VOORBEELD

```
procedure verwissel(name x, y: integer);
    begin x := x + y; y := x - y; x := x - y end
```

Als deze procedure wordt aangeroepen als verwissel(a, b), waarbij a en b waarden hebben, dan worden de waarden van a en b inderdaad verwisseld. Wordt de procedure aangeroepen met verwissel(t, t) dan zal t na afloop de waarde 0 hebben, omdat in de rechterleden van de laatste assignments in feite staat t - t.
(einde voorbeeld)

Een taal als EUCLID verbiedt dan ook dat dezelfde actuele parameter wordt gebruikt bij verschillende formele parameters.

In § 7.3 hebben we al over een constructie, de exit, gesproken om een repetitie te beëindigen als een bepaalde situatie optreedt. In feite is dit een gecontroleerde wijze van gebruik van een sprongopdracht. We zullen nu nog twee constructies bekijken die eenzelfde soort effect hebben. De eerste constructie is de zogenaamde *exception* in Ada (die ook reeds in § 7.3 aan de orde kwam). In Ada bestaat de mogelijkheid om na een declaratie van een exception deze exception te gebruiken (the raise of the exception).
Bijvoorbeeld:

<u>if</u> determinant = 0 <u>then</u> <u>raise</u> singular <u>end</u> <u>if</u>

Deze uitzonderingstoestand wordt dan afgehandeld door een apart stukje programma (aangeduid door singular) dat de programmeur heeft geplaatst aan het einde van het subprogramma waarin deze exception voorkomt. Het is dus een sprong naar het einde van het subprogramma met eventueel een aantal extra acties, door de programmeur opgegeven.

De tweede constructie is eigenlijk een verbijzondering van de eerste. In een aantal talen is het mogelijk zogenaamde *assertions* of assert-statements in een programma op te nemen. Dit zijn in feite boolean expressies. Het zijn beschrijvingen van de toestand die op dat moment in het proces moet gelden. Levert de expressie de waarde true op, dan wordt gewoon de volgende statement uitgevoerd, zo niet dan treedt een uitzonderingstoestand op die ook weer zou kunnen inhouden dat een apart stuk programma wordt uitgevoerd en/of dat verdere verwerking wordt gestaakt.

8 BESPREKING VAN PASCAL, FORTRAN EN COBOL

8.1 INLEIDING

In het volgende zullen elementen en constructies worden genoemd van de talen Pascal, FORTRAN en COBOL. Er wordt niet getracht volledig te zijn, maar er is naar gestreefd de karakteristieken van elk van de talen te belichten. De elementen en constructies die in hoofdstuk 7 aan de orde kwamen zullen hier, voorzover aanwezig, per taal behandeld worden. Mogelijkheden voor parallelle programmering zijn in geen van de drie talen aanwezig.
Voor Pascal wordt het Pascal Report gevolgd.
Voor FORTRAN wordt de standaardversie FORTRAN 77 gevolgd.
Voor COBOL wordt de 1974 AMERICAN NATIONAL STANDARD COBOL versie gevolgd.

8.2 DE STRUCTUUR VAN EEN PROGRAMMA

Pascal

Een programma in Pascal wordt formeel gedefinieerd door:

<program> ::= <heading> <block>.
<heading> ::= program <identifier> [(<identifier list>)];
<block> ::= {<declaration>;} <compound statement>
<compound statement> ::= begin <statement> {;<statement>} end

De eventuele <identifier list> in de program <heading> specificeert de files via welke het programma invoer en uitvoer pleegt (zie § 8.10).

De declaraties zijn verdeeld in drie groepen:
- constanten en typen,
- variabelen,
- procedures en functies.

De <compound statement> is een speciale vorm van een statement, zodat binnen een <compound statement> ook weer <compound statements> kunnen voorkomen. Omdat in een <compound statement> echter geen declaraties voorkomen, is de enig mogelijke blok structuur die binnen procedures en functies:

<procedure declaration> ::= <procedure heading> ; <block>;
<function declaration> ::= <function heading> ; <block>;

Door deze blokstructuur is de structuur van een Pascal-programma recursief. De geheugentoewijzing is dan ook dynamisch: op het moment dat een programma/procedure/functie aangeroepen wordt, wordt de benodigde geheugenruimte gereserveerd, bij afloop van het programma of de procedure/functie wordt de gereserveerde geheugenruimte (die dan niet meer nodig is!) teruggegeven aan het systeem. Indien een procedure/functie meer dan één keer aangeroepen wordt, wordt elke keer opnieuw geheugenruimte gereserveerd; de lokale variabelen kunnen dus niet een waarde behouden.

Voor de 'scope-rules' voor variabelen, procedures en functies verwijzen we naar § 7.4; de 'scope-rules' van Pascal zijn gelijk aan die van ALGOL 60.

Het statisch einde van een programma (het symbool end van het program <block>) is ook het dynamisch einde; dit geldt ook voor procedures en functies.

De programmatekst is een rij van symbolen, die sequentieel gelezen wordt. Overgang op een nieuwe regel en het gebruik van spaties hebben geen enkele consequentie of betekenis; aldus kan de structuur van het programma in de lay-out tot uiting komen.

FORTRAN

Een gedeelte van de grammatica is:

<program> ::= <main program> {<subprogram>}
<main program> ::= <heading> <body>
<heading> ::= {<declaration>}
<body> ::= {<statement>}$_1$ END
<subprogram> ::= <subprogram heading> <head> <body>
<subprogram heading> ::= <subroutine>
|<function>
|<block data>

Een programma in FORTRAN bestaat dus uit een hoofdprogramma (<main program>), eventueel gevolgd door een of meer subprogramma's. De <declarations> zijn van variabelen en functies; ook de definitie van globale variabelen (COMMON, zie § 8.8) en equivalente variabelen (EQUIVALENCE, § 8.8) hoort hierbij. Voor de statements geldt:

<statement> ::= <format statement>
|<subprogram call>
|<assignment statement>
|<control statement>
|<input-output statement>
|<dynamic end statement>
|<data initialisation>

De eventuele <data initialisation> moet voorafgaan aan de andere statements. Met de <format statement> kan de lay-out van in- of uitvoer worden vastgelegd (zie § 8.10). De <control statements> zijn de DO, IF en GOTO statements.

Een <block data> subprogramma dient voor de declaratie en de initialisatie van COMMON variabelen.

De structuur van een FORTRAN-programma is niet recursief: een programma bestaat uit een hoofdprogramma en subprogramma's, die alle uit declaraties en statements bestaan. De verwerking van een FORTRAN-programma begint met een (statische) toewijzing van geheugen voor alle declaraties in hoofdprogramma en subprogramma's. De geheugentoewijzing, voor elk van de delen, wordt teniet gedaan bij afloop van het gehele programma. De lokale variabelen van een subprogramma blijven dus bestaan. Indien een subprogramma meer dan één keer aangeroepen wordt, geldt volgens de definitie echter steeds dat de lokale variabelen niet geïnitialiseerd zijn; in veel implementaties blijven toegekende waarden echter bestaan, zodat daar bij een volgende aanroep (eventueel) gebruik van gemaakt kan worden.

Het statisch einde van een programma (tekst) wordt aangegeven door middel van het keyword END. Met de <dynamic end statement> STOP (voor het hoofdprogramma) en RETURN (voor subprogramma's) wordt het dynamisch einde aangegeven.

FORTRAN is erg kaart-georiënteerd en er zijn strikte regels voor de opmaak van de regels van een programmatekst:

a. De kolommen 1-5 kunnen een eventuele label bevatten.
b. Elke statement begint op een nieuwe regel; indien een statement niet op één regel past, kan op de volgende regel worden verder gegaan door daar in kolom 6 een symbool, anders dan 0 of spatie, te plaatsen.
c. De kolommen 7-72 bevatten de eigenlijke tekst.
d. Een letter c of het teken * in kolom 1 betekent dat deze regel in kolom 2-72 commentaar bevat.
e. De kolommen 73-80 kunnen voor identificatie (bijvoorbeeld een volgnummer) gebruikt worden.

COBOL

Een COBOL-programma bestaat uit vier divisies, achtereenvolgens: IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, DATA DIVISION en PROCEDURE DIVISION. Deze divisies worden onderverdeeld in secties en deze laatste weer in paragrafen (niet bij de DATA DIVISION). Per divisie zijn soms enkele secties en paragrafen verplicht.

De structuur van de IDENTIFICATION DIVISION is:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. <program name>.
<other information, e.g. date>
```

De programma-naam moet met een letter beginnen en gevolgd worden door een punt.

De ENVIRONMENT DIVISION luidt:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. (source)computer-name.
OBJECT-COMPUTER. (object)computer-name.
INPUT-OUTPUT-SECTION.
FILE-CONTROL.
```

De CONFIGURATION SECTION beschrijft in aparte paragrafen de machine die het programma vertaalt (source computer) en de machine die het vertaalde programma verwerkt (object computer).

In de paragraaf FILE-CONTROL volgt nu een beschrijving van de invoer- en uitvoerapparatuur via welke met files, die met naam worden genoemd, gewerkt gaat worden.

In de DATA DIVISION worden de datastructuren en grootheden gedeclareerd. De DATA DIVISION is verdeeld in SECTIONS en de drie voornaamste secties beschrijven files (met de records), de WORKING STORAGE (voor datastructuren en grootheden) en constanten.

De PROCEDURE DIVISION geeft de eigenlijke procesbeschrijving. Ook de PROCEDURE DIVISION is onderverdeeld in SECTIONS, bestaande uit één of meer paragrafen. Een paragraaf begint met de naam ervan en bestaat voorts uit één of meer 'sentences' (zinnen). Een sentence bestaat gewoonlijk uit een aantal statements en eindigt met een punt. De statements binnen een zin worden gescheiden door één of meer spaties eventueel met een puntkomma. Een sentence mag over regelgrenzen gaan; een regel mag meer dan één sentence bevatten. In afwijking tot de andere divisies zijn in de PROCEDURE DIVISION de namen van de secties vrij te kiezen; zo'n naam wordt altijd gevolgd door het woord SECTION.

De structuur van een COBOL-programma is statisch, niet recursief. Lokale variabelen bestaan niet. Alle gebruikte datastructuren en grootheden worden in de DATA DIVISION vastgelegd. De geheugentoewijzing is dus ook statisch.

Het dynamisch einde van een programma wordt aangegeven door STOP RUN. Deze statement hoeft niet perse de laatste van de programmatekst te zijn en mag meer dan één keer voorkomen.

Ook COBOL is kaartgeoriënteerd:
a. De kolommen 1-6 zijn bestemd voor nummering.
b. Kolom 7 is blanco, tenzij de regel een voortzetting is van de vorige (dan symbool -) of tenzij de regel commentaar bevat (dan symbool *).
c. De naam van een divisie, sectie en paragraaf begint in één van de kolommen 8, 9, 10 of 11.
d. Sentences staan in de kolommen 12-72.
e. De kolommen 73-80 worden voor identificatie gebruikt (bijvoorbeeld een volgnummer).

8.3 BASISSYMBOLEN

Elke taal maakt gebruik van een verzameling basissymbolen, ofwel eindsymbolen (terminals), zoals we in hoofdstuk 3 zagen. De drie talen die we hier bespreken hebben alle een verzameling basissymbolen die weer te geven is als:

<basic symbol > ::= <letter> | <digit> | <special symbol>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Pascal

In Pascal zijn zowel hoofdletters als kleine letters toegestaan. Veel implementaties staan echter alleen hoofdletters toe. De verzameling <special symbols> bestaat uit aritmetische operatoren (+, -, *, /, div, mod), relationele operatoren (=, <=, < , >=, >, <>), logische operatoren (or, and, not), speciale karakters ( ], [, (,), ', :=, ., ,, :, ;, ..) en een aantal woorden met een zeer specifieke betekenis (bijvoorbeeld begin, end, while, do, record, set, function, procedure, type, etc.). Om deze woorden te onderscheiden van identifiers worden ze onderstreept (evenals div, mod, or, and, not); in sommige boeken worden ze vet gedrukt, bijvoorbeeld **begin**. In veel implementaties zijn het 'reserved words' die niet als naam mogen voorkomen; in dit geval is dan geen onderstreping of een andere vorm van onderscheid nodig.

Verder is er de verzameling 'standard identifiers' (bijvoorbeeld integer, real, abs, exp, true, false) die ook elk een specifieke betekenis hebben. Deze mogen – in tegenstelling tot 'reserved words' – wèl als naam gebruikt worden, maar dan verliest zo'n identifier de oorspronkelijke betekenis!. Dit is erg onduidelijk en is dus af te raden.

FORTRAN

In FORTRAN zijn alleen hoofdletters toegestaan. De verzameling <special symbols> bestaat uit aritmetische operatoren (+, -, *, /, **), logische waarden (.TRUE., .FALSE.), relationele operatoren (.LT., .EQ., .LE., .NE., .GE., .GT.), logische operatoren (.AND., .OR., .NOT.) en nog enkele speciale karakters (=, (,), *, :, ., ,, $). Hier worden de woorden die als <special symbol> optreden onderscheiden van namen door ze door twee punten te omsluiten. Voor andere woorden met een speciale betekenis (bijvoorbeeld INTEGER, COMPLEX, COMMON, SUBROUTINE), de zogenaamde 'keywords', geldt dat zo'n woord gerust als naam van een variabele gebruikt mag worden; de actuele betekenis van zo'n woord blijkt bij verwerking uit de context.

COBOL

Ook in COBOL zijn alleen hoofdletters toegestaan. De verzameling <special symbols> bestaat uit aritmetische operatoren (+, -, *, **, /), relationele operatoren (<, >) en speciale karakters ($, ", ,, ., (,), ;). Er is een grote verzameling 'reserved words' (bijvoorbeeld AND, CALL, IF, EQUAL, PROCEDURE, MOVE), elk met een zeer specifieke betekenis. Deze 'reserved words' mogen niet als namen van grootheden gebruikt worden.

8.4 DATATYPEN; VARIABELEN EN DECLARATIES

Pascal

Variabelen, typen en procedures/functies hebben een naam, die bestaat uit een letter, gevolgd door nul of meer cijfers of letters. Er is geen beperking van het aantal karakters, maar de meeste implementaties zullen wel een bovengrens stellen.

Pascal kent de enkelvoudige typen integer, real, char en boolean. Als waarden kunnen optreden gehele getallen, reële getallen met de letter E als 10-macht (schaalfactor), de karakters, en de logische waarden true en false. Tevens zijn waarden van het type string toegestaan; dit is echter geen type, en er kunnen geen variabelen van het type string gebruikt worden.

Naast de standaard enkelvoudige typen bestaat de mogelijkheid opsommingen en subranges als type te gebruiken (zie hoofdstuk 7).

Introductie van typen gebeurt met een 'type-definition', waarbij het type een naam krijgt.
Bijvoorbeeld:

```
type dag = (zondag, maandag, dinsdag, woensdag, donderdag,
         vrijdag, zaterdag);
     maand = 1..12;
     werkdag = maandag..vrijdag;
```

Alle variabelen moeten gedeclareerd worden, waarbij een naam en een type genoemd worden.
Bijvoorbeeld:

```
var i, j : integer;
      m: maand;
    dag : werkdag;
```

Ook aan constanten kunnen namen worden toegekend.
Bijvoorbeeld:

```
const pi = 3.1415926;
      imax = 10000;
```

Pascal kent ook het type pointer. Dit type komt geheel overeen met de pointers zoals beschreven in hoofdstuk 7.

Tevens kent Pascal de set als enkelvoudig type (zie hoofdstuk 7). De declaratie van twee variabelen d en w van het type set van werkdagen is als volgt:

```
var d, w: set of werkdag;
```

Operaties op sets zijn:
- test of een element in de waarde voorkomt, bijvoorbeeld: maandag in d
- deelverzameling testen, bijvoorbeeld: w <= d
- superset testen, bijvoorbeeld: d >= w
- vereniging, bijvoorbeeld: d + w
- doorsnede, bijvoorbeeld: d * w
- verschil, bijvoorbeeld: d - w

Als samengestelde typen kent Pascal arrays, records en files. Een array mag van willekeurige dimensie zijn (≧ 1) zijn. Voor iedere dimensie wordt bij declaratie een index-type gespecificeerd.
Bijvoorbeeld:

```
var x: array [1..500, 10..15] of real;
    tabel: array [dag] of integer;
```

Als index-type mag elk enkelvoudig (gedefinieerd) type optreden behalve real. Als component type mag elk (gedefinieerd) type optreden, dus ook een array.
Bijvoorbeeld:

```
var A: array [1..10] of array of [1..100] of boolean;
```

De array-componenten worden benoemd door een index, die een variabele, een constante of een expressie mag zijn.
Bijvoorbeeld:

```
x[1, 12]
tabel[dinsdag]
A[i][50]
A[i][10 * (i-1) + i]
```

Bij sommige implementaties van Pascal worden A[i][50] en A[i, 50] als equivalent beschouwd; in de definitie van Pascal is dat echter niet zo!

Een record-structuur mag uit een willekeurig groot aantal velden bestaan. Bij declaratie wordt voor elk veld de naam en het type vastgelegd (dat mag ook een samengesteld type zijn).
Bijvoorbeeld:

```
type datum = record d: 1..31;
                    m: maand;
                    j: 1900..2000
             end;
var dat: datum;
```

De velden kunnen geselecteerd worden door het specificeren van de naam.
Bijvoorbeeld:

```
dat := (4,2,1981); heeft hetzelfde effect als de drie statements:
dat.d := 4;
dat.m := 2;
dat.j := 1981;
```

Met behulp van de with-statement kunnen de velden alle benoemd worden zonder de naam van de variabele te gebruiken.
Bijvoorbeeld:

```
with dat do begin d := 4; m := 2; j := 1981 end
```

is equivalent met de drie assignment statements hierboven.

Bij declaratie van een record type mag voor een veld ook een keuze uit meerdere typen gespecificeerd worden; de waarde van het veld moet dan altijd van één van de genoemde typen zijn. We spreken van 'variante records'. Welk type steeds actueel is hangt af van de waarde van de zogenaamde 'tag'.

VOORBEELD

```
type paginasoort = (eerste, volgende);
type pagina = record case p: paginasoort of
                       eerste: (titel: array [1..50] of char;
                                auteur: [1..50] of char;
                                ISBN: array [1..10] of 0..9);
                       volgende: (tekst: array [1..100]
                                         of array [1..80]
                                         of char)
                     end
(p is de tag)
Waarde-toekenning:
var pag: pagina;
pag.p := eerste;
pag.titel := ...; pag.auteur := ...; pag.ISBN := ...;
```

(einde voorbeeld)

Pascal kent ook de file-structuur die overeenkomt met de sequence structuur uit hoofdstuk 7: een rij waarden van een bepaald type. Steeds is slechts één waarde toegankelijk; de andere waarden kunnen door een sequentiële verwerking toegankelijk worden. Het aantal waarden is onbepaald. Bij declaratie moet het type gespecificeerd worden.
Bijvoorbeeld:

```
type F = file of integer;
     G = file of record re, im: real end;
var s: F;
    g: G;
```

Bij elke file hoort een wijzer (voor een file f geven we die aan met f↑) die steeds de als volgende te verwerken waarde aanwijst.
De eenvoudigste operaties zijn:
- read(f, x) met als effect dat x de aangewezen waarde krijgt en f↑ naar de volgende waarde gaat wijzen;
- eof(f), een boolean functie, die true oplevert als f↑ voorbij de laatste waarde van de rij wijst, en anders false;
- write(f, x) met als effect dat – mits eof(f) true oplevert – de waarde van x achter aan de rij wordt toegevoegd, en dat eof(f) true blijft;
- reset(f) met als effect dat f↑ naar de eerste van de rij waarden gaat wijzen;
- rewrite(f) met als effect dat de rij leeg wordt, en dus eof(f) true.

Er zijn nog andere operaties, onder andere op de wijzer f↑, die we hier echter niet behandelen.

VOORBEELD

```
   reset(s); read(s)
   rewrite(g); write(g, r)
```
(einde voorbeeld)

In Pascal bestaat ook een standaard type

```
   text = file of char;
```

De waarde van een 'textfile' is een rij waarden van het type char; deze rij is echter onderverdeeld in een rij regels, zodat er extra operaties nodig zijn, bijvoorbeeld voor overgang op een nieuwe regel, en extra tests of een regel als geheel gebruikt is. Invoer en uitvoer gebeurt met de standaard 'textfiles' input en output; in § 8.10 zullen we de specifieke I/O-operaties op 'textfiles' behandelen.

FORTRAN

Ook in FORTRAN wordt een naam aangegeven door een identifier, nu bestaande uit maximaal zes letters en/of cijfers en beginnend met een letter.

FORTRAN kent als enkelvoudige typen INTEGER, REAL, DOUBLE PRECISION, COMPLEX en LOGICAL. Daarnaast heeft men nog de beschikking over het CHARACTER-datatype, voor het werken met teksten van vaste lengte. Als een naam van een enkelvoudige variabele begint met een I, J, K, L, M of N en deze moet van het type INTEGER zijn, dan behoeft deze niet te worden gedeclareerd. Ook variabelen van het type REAL behoeven niet te worden gedeclareerd: de naam begint dan met een letter ongelijk aan de hierboven genoemde.

Variabelen mogen echter ook expliciet worden gedeclareerd in een zogenaamde 'type statement' (en voor de duidelijkheid is dit ook aan te bevelen). Variabelen van het type LOGICAL, COMPLEX, DOUBLE PRECISION en CHARACTER moeten worden gedeclareerd. Enkele voorbeelden (bij de CHARACTER-declaratie geven *50 en *35 de lengte aan):

```
   REAL LOON, NUM, T(10)
   INTEGER SAL, BELAS, I(10, 12, 10)
   LOGICAL WAAR, P(10, 15)
   COMPLEX Z
   DOUBLE PRECISION DP, AA
   CHARACTER *50 C(25)
   CHARACTER *35 T1, T2, R1
```

Met de IMPLICIT-statement kan aangegeven worden dat alle variabelen, waarvan de naam met een bepaalde letter begint, tot een bepaald type behoren. Bijvoorbeeld:

IMPLICIT INTEGER (X)

geeft aan dat alle variabelen, waarvan de naam met een X begint, van het type integer zijn.

Een array heeft in FORTRAN maximaal 7 dimensies. De ondergrens is in ieder van de dimensies standaard 1 en behoeft bij de declaratie niet te worden aangegeven; de bovengrens wordt aangegeven door een numerieke constante of door een numerieke variabele (dit laatste is alleen toegestaan in een subroutine). Een ondergrens (bijvoorbeeld anders dan 1) mag wel bij de declaratie aangegeven worden. Zoals we reeds hebben gezien kunnen enkelvoudige variabelen en arrays worden gedeclareerd in een type statement, een array kan ook worden gedefinieerd door een dimension statement:

DIMENSION P(5), Q(10, 10), L(25)

De indices van een array mogen constanten, enkelvoudige variabelen of expressies van het type INTEGER zijn.

Naast de array-structuur kent FORTRAN de structuur COMPLEX, waarvan de waarden bestaan uit geordende paren getallen van het type REAL (reëel en imaginair deel van het complexe getal).

FORTRAN kent als (constante) waarden de getallen, de logische waarden .TRUE. en .FALSE. en teksten (rijen karakters tussen apostroffen, bijvoorbeeld 'TEKST'). De getallen kunnen van het type INTEGER en het type REAL zijn. In FORTRAN is 5. een getal van het type REAL; voor de 10-macht als schaalfactor gebruikt men de letter E.

COBOL

Identifiers bestaan uit rijen letters, cijfers en het streepje (-), waarbij het streepje niet het eerste en het laatste teken van de naam mag zijn en de naam ten minste één letter moet bevatten. De naam bestaat uit hoogstens 30 tekens.

Alle gebruikte namen voor enkelvoudige en gestructureerde variabelen moeten in de DATA DIVISION gedeclareerd worden. Beschikbare 'datatypen' zijn:

- elementair type met numerieke waarde
- elementair type met alfanumerieke waarde (dit is een tekst waarvan de karakters behoren tot de basissymbolen)
- elementair type met alfabetische waarde (dit is een tekst, waarvan de karakters behoren tot een subset van de set basissymbolen, te weten de letters A tot en met Z en het speciale teken -)
- gestructureerde type record
- gestructureerde type array.

COBOL kent niet expliciet verschillende elementaire datatypen; de type-indicatie en het formaat van een variabele worden echter aangegeven met een zogenaamde PICTURE. Hierin wordt aangegeven uit hoeveel tekens de waarden van de variabelen mogen bestaan en of het gaat om numerieke waarden (9-picture), alfanumerieke waarden (X-picture) of zuiver alfabetische waarden (A-picture). Als bijvoorbeeld een variabele numerieke waarden kan aannemen, bestaande uit vijf cijfers, wordt na de naam van de variabele in de DATA DIVISION opgegeven PICTURE 99999 of PICTURE 9(5).

Constanten in COBOL worden verdeeld in 'numeric literals', 'non-numeric literals' en 'figurative constants' (ofwel getallen, teksten en constanten die een naam hebben gekregen zoals ZERO en SPACE).

De naam van een variabele wordt gekenmerkt door:
- voor een enkelvoudige variabele: identifier
- voor een array-element : identifier, gevolgd door 1, 2 of 3 indices tussen ronde haken
- voor een record-component : identifier, eventueel gevolgd door de naam (namen) van de omvattende component(en).

Declaratie van de variabelen:
1. Elementaire variabelen worden in de WORKING STORAGE SECTION gedeclareerd.
   De declaratie bestaat uit:
   - levelnummer (=77)
   - naam
   - PICTURE, zijnde: 9-PICTURE voor een numerieke waarde
     A-PICTURE voor een alfabetische waarde
     X-PICTURE voor een alfanumerieke waarde

   Het aantal in beslag genomen posities wordt aangeduid door dit aantal tussen haken op te geven òf de PICTURE-aanduiding zó vaak te herhalen als er posities zijn.

   Bijvoorbeeld:

```
77 veld-1     PICTURE 999.    numerieke waarde van 3 cijfers
77 veld-2     PICTURE A(5).   alfabetische waarde van 5 letters
77 veld-3     PICTURE X(10).  alfanumerieke waarde van 10 karak-
                              ters.
```

Een beginwaarde toekennen aan een veld is mogelijk door er een VALUE (waarde) aan toe te voegen, zoals:

```
77 CONSTANTE-0      PICTURE 9     VALUE 0.
77 CONSTANTE-PIET   PICTURE A(4)  VALUE "PIET".
```

2. Variabelen van het type record worden in de FILE SECTION gedeclareerd.
   De declaratie bestaat uit:
   - levelnummer (01 tot en met 49)
   - naam
   - PICTURE (alleen voor het elementaire type binnen het record)

VOORBEELD

```
 01 DAGCODE.
    02 DAG          PICTURE 99.
    02 MAAND        PICTURE 99.
    02 JAAR         PICTURE 9(4).
```

(einde voorbeeld)

VOORBEELD

```
 01 PERSOONSGEGEVENS.
    05 NAAM.
       10 ACHTERNAAM              PICTURE X(20).
       10 VOORNAMEN.
          15 EERSTE-VOLUIT        PICTURE X(10).
          15 REST-INITIALEN       PICTURE X(6).
    05 GEBOORTEDATUM.
       10 JAAR                    PICTURE 9(4).
       10 MAAND                   PICTURE 9(2).
       10 DAG                     PICTURE 9(2).
    05 SALARISNUMMER              PICTURE 9(6).
    05 DATUM-IN-DIENST.
       10 JAAR                    PICTURE 9(4).
       10 MAAND                   PICTURE 9(2).
       10 DAG                     PICTURE 9(2).
    05 BURGERLIJKE STAAT.
       10 GEHUWD                  PICTURE A.
       10 AANTAL KINDEREN         PICTURE 9(2).
    05 FUNCTIE                    PICTURE X(10).
```

(einde voorbeeld)

Bij de recordbeschrijvingen wordt gebruik gemaakt van niveaus (of levels), waarin het record verdeeld wordt; deze levels worden genummerd, waarbij 01 gebruikt wordt voor het record en de opvolgende levels een hoger nummer krijgen (dat niet noodzakelijk aansluitend behoeft te zijn). Voor de levels worden de nummers 01 tot en met 49 gebruikt. De levels worden in de regel verder naar rechts geschreven. Als een level niet meer onderverdeeld wordt in onderlevels moet door een PICTURE de lay-out aangegeven worden van de desbetreffende component.

Elke component van een record moet eenduidig geïdentificeerd kunnen worden. Dit gebeurt met de naam van de component. Hebben verscheidene componenten dezelfde naam, dan dient ook de naam van de omvattende component(en) opgenomen te worden.

In het record PERSOONSGEGEVENS kan aan het jaar van indiensttreding gerefereerd worden door:

JAAR OF DATUM-IN-DIENST òf
JAAR OF DATUM-IN-DIENST OF PERSOONSGEGEVENS

We moeten met een recordbeschrijving bijvoorbeeld ook de lay-out

van een ponskaart (als record) kunnen vastleggen. Dat betekent dat we ook niet-gebruikte kolommen moeten kunnen beschrijven. Dit gebeurt door een zogenaamde FILLER te introduceren:

```
01 DATUM.
   02 DAG            PICTURE 99.
   02 FILLER         PICTURE X(8).
   02 MAAND          PICTURE 99.
   02 FILLER         PICTURE X(8).
   02 JAAR           PICTURE 9(4).
   02 FILLER         PICTURE X(54).
   02 VOLGNUMMER     PICTURE 99.
```

De programmeur kan natuurlijk ook andere namen aan deze posities geven. Het level FILLER mag niet verdeeld worden in onderlevels; in de PROCEDURE DIVISION kan niet aan FILLER gerefereerd worden.

Records waarvan de structuur met een recordbeschrijving is vastgelegd zullen in het algemeen in het programma een waarde krijgen om vervolgens in een bestand (file) te worden opgeslagen of krijgen hun waarde door ze te lezen van een bestaand bestand. Omdat dit bestand een fysiek bestaand geheel is moet behalve de logische structuur van het record ook iets worden aangegeven over de fysieke structuur van het bestand waarin de records zijn of worden opgeslagen. Dit gebeurt met behulp van een FD-statement (file-description statement) dat direct vooraf moet gaan aan de beschrijving van de recordstructuur. Met behulp van dit statement kunnen veel attributen van het bestand worden opgegeven waarvan alleen de drie belangrijkste hier worden vermeld.

De vorm van het statement is:

```
FD naam van het bestand
 BLOCK CONTAINS ..... RECORDS
 RECORD CONTAINS ..... CHARACTERS
 DATA RECORD IS naam van record.
01 naam van record.
   rest van recordbeschrijving
```

Met BLOCK CONTAINS .... geeft men aan hoeveel logische records in één fysiek record (block) moeten worden opgenomen.

Met RECORD CONTAINS .... wordt aangegeven hoeveel posities het record in beslag neemt. Aan de hand van de PICTURE-definitie kan dit eenvoudig worden bepaald. Zo bestaat het record PERSOONSGEGEVENS uit 71 karakters.

DATA RECORD IS .... tenslotte geeft aan welk logisch record hoort bij de met de naam van het bestand aangegeven file.

In bovenstaand voorbeeld zou het FD-statement er uit kunnen zien als:

```
FD GEBOORTEDATA
 BLOCK CONTAINS 10 RECORDS
 RECORD CONTAINS 80 CHARACTERS
 DATA RECORD IS DATUM.
```

3. Variabelen met een array-structuur worden in de WORKING STORAGE SECTION gedeclareerd.
   De declaratie bestaat uit:

   - levelnummer (01 tot en met 49)
   - naam
   - OCCURS .... TIMES
   - PICTURE (alleen voor het elementaire type binnen het array).

Ook bij arrays wordt gebruik gemaakt van niveaus; deze niveaus worden genummerd, waarbij 01 gebruikt wordt voor het array en de volgende niveaus een hoger nummer krijgen.

Een array mag 1-, 2- of 3-dimensionaal zijn.

Een array met de naam 'MATRIX' bestaande uit drie rijen en vijf kolommen wordt als volgt gedeclareerd:

```
01 MATRIX.
   03 RY OCCURS 3 TIMES.
      05 ELEMENT OCCURS 5 TIMES PICTURE 99.
```

De individuele elementen kunnen nu aangegeven worden als ELEMENT(M,N), waarbij M en N namen van numerieke variabelen of constanten zijn voor respectievelijk de rij- en de kolomwaarde.
Een array mag optreden als item in een record of
een record mag optreden als item in een array.

Een voorbeeld van dit laatste:

```
01 ADRESLYST.
   02 EENHEID OCCURS 100 TIMES.
      03 NAAM PICTURE X(36).
      03 STRAAT PICTURE X(30).
      03 WOONPLAATS PICTURE X(30).
```

Men kan nu werken met EENHEID(M), drie gegevens inhoudende, en bijvoorbeeld met STRAAT(M).

Ook kan men tabellen hebben als:

```
01 PRODUCTLYST
   02 PRODUCT1 OCCURS 50 TIMES PICTURE 9(6).
   02 PRODUCT2 OCCURS 50 TIMES PICTURE 9(6).
   02 PRODUCT3 OCCURS 50 TIMES PICTURE 9(6).
```

Hierbij kan men de tabel opgedeeld denken uit eerst 50 codes voor de produktsoort1, dan 50 codes voor produktsoort 2 en tenslotte 50 codes voor produktsoort 3; men zou kunnen zeggen dat de drie ééndimensionale tabellen PRODUCT1, PRODUCT2 en PRODUCT3 zijn samengevoegd tot één tabel. Voor een tabel is altijd een naam nodig en daarna zijn nog namen nodig voor ieder van de dimensies; deze namen kunnen in de PROCEDURE DIVISION gebruikt worden, zodat de tabel op verschillende niveaus benaderd kan worden.

8.5 EXPRESSIES EN ASSIGNMENT

Voor elk van de drie talen geven we aan hoe aritmetische en logische expressies opgebouwd kunnen worden. Vervolgens bespreken we hoe expressies gebruikt kunnen worden bij waarde-toekenning. Andere vormen van gebruik van expressies komen verderop aan de orde.
Op p.154 geven we in tabelvorm de operatoren voor elk van de talen.

Voor de evaluatie van de expressie is het nodig de prioriteiten van de gebruikte operatoren te kennen. Als we prioriteiten toekennen gebeurt de waardeberekening als volgt:
- Als alle operatoren gelijke prioriteit hebben worden ze van links naar rechts uitgevoerd (dit geldt niet voor FORTRAN!).
- Indien operatoren met verschillende prioriteiten voorkomen, worden eerst de operatoren met de hoogste prioriteit uitgevoerd, vervolgens die met de op één na hoogste prioriteit, enzovoorts.
- Door middel van ronde haakjes zijn beide voorgaande regels te omzeilen: de waarde van de expressie binnen de haakjes wordt eerst berekend (volgens deze regels!).

Pascal

De prioriteiten van de operatoren zijn in Pascal als volgt:

prioriteit	operator
4	not
3	* / div mod and
2	+ - or
1	= <> > >= < <= in

Standaard-functies, die in expressies gebruikt mogen worden, zijn bijvoorbeeld:
- aritmetisch: abs, sqr (kwadraat), sin, cos, arctan, exp, ln, sqrt;
- logisch: odd (test op oneven), eoln (textfiles), eof;
- opsomming: succ (de opvolger in de opsomming), pred (de voorganger in de opsomming), ord (de plaats in de opsomming; telling begint bij 0);
- karakters: ord (de plaats in de rij karakters), chr (het karakter op de plaats ...).

De assignment statement heeft in Pascal de vorm:

<variabele> := <expression>

De variabele en de expresie moeten van hetzelfde type zijn, met als uitzonderingen:

	Pascal	FORTRAN	COBOL
aritmetische operatoren			1)
- machtsverheffing		**	**
- vermenigvuldiging	*	*	*
- deling	/	/	/
- gehele deling	div, mod		
- optelling	+	+	+
- aftrekking	-	-	-
relatie-operatoren			2)
- kleiner	<	.LT.	IS LESS THAN IS <
- kleiner of gelijk	<=	.LE.	IS NOT GREATER THAN IS NOT >
- gelijk	=	.EQ.	IS EQUAL TO IS =
- niet gelijk	<>	.NE.	IS NOT EQUAL TO IS NOT =
- groter of gelijk	>=	.GE.	IS NOT LESS THAN IS NOT <
- groter	>	.GT.	IS GREATER THAN IS >
relatie-operatoren op sets			
- gelijk	=		
- niet gelijk	<>		
- deelverzameling	<=		
- superset	>=		
- element van	in		
logische operatoren			
- ontkenning	not	.NOT.	NOT
- conjunctie	and	.AND.	AND
- disjunctie	or	.OR.	OR
- implicatie	<=		
- equivalentie	=		
- exclusive or	<>		

1) Voor optellen, aftrekken, vermenigvuldigen en delen zijn in COBOL ook andere constructies mogelijk dan in de vorm van aritmetische expressies.

2) In COBOL zijn meerdere aanduidingen voor een relatie mogelijk; we geven steeds de alternatieven.

- als de expressie een variabele van een subrange type is mag de variabele van het oorspronkelijke type zijn;
- aan een variabele van een subrange type mag een waarde uit de waardenverzameling van het oorspronkelijke type toegekend worden mits die waarde in het juiste interval ligt;
- aan een real variabele mag een integer waarde toegekend worden.

FORTRAN

De prioriteiten voor aritmetische operatoren zijn in FORTRAN:

prioriteit	operator
3	**
2	* /
1	+ -

Bij gelijkheid van operatoren tijdens de evaluatie geldt:
- operatoren met prioriteit 3 worden van rechts naar links uitgevoerd (2**3**4 wordt berekend als 2**(3**4));
- operatoren met prioriteit 2 worden van links naar rechts uitgevoerd;
- operatoren met prioriteit 1 worden van links naar rechts uitgevoerd.

Voor de evaluatie van logische expressies geldt:
- aritmetische operatoren hebben de hoogste prioriteit;
- relatie-operatoren hebben de op één na hoogste prioriteit;
- de logische operator .NOT. heeft de op twee na laagste prioriteit;
- de logische operator .AND. heeft de op één na laagste prioriteit;
- de logische operator .OR. heeft de laagste prioriteit;
- bij gelijke prioriteit worden de expressies van links naar rechts uitgewerkt.

Van de standaard-functies die in expressies te gebruiken zijn noemen we enkele aritmetische:
EXP, ALOG, ALOG10, SIN, COS, TAN, SQRT, ABS, MOD, etc.

Op de verzameling basissymbolen is een volgorde gedefinieerd, zodat op teksten ook logische operaties mogelijk zijn; bijvoorbeeld:

```
'AAP'.LT.'AAR'
```

De assignment statement heeft in FORTRAN de vorm:

```
<variabele> = <expression>
```

Als de variabele van het type LOGICAL is, moet de expressie logisch zijn. Als de variabele van het type CHARACTER is, moet de expressie ook van dat type zijn (een tekst, al dan niet opgebouwd door middel van standaardfuncties voor het type CHARACTER). Als

de variabele van een aritmetisch type is, moet de expressie ook aritmetisch zijn, waarbij geldt:
- als een REAL of DOUBLE PRECISION waarde aan een INTEGER variabele wordt toegekend, wordt afgekapt (7.7 wordt 7);
- een INTEGER waarde mag aan een REAL of DOUBLE PRECISION variabele worden toegekend;
- bij deling van twee INTEGERs wordt afgekapt; het resultaat is INTEGER (7/3 wordt 2).

COBOL

Van de expressies bestaan twee soorten:

- aritmetische expressie:
  Deze is gelijk aan de eenvoudige aritmetische expressie in Pascal. Daar COBOL niet in de eerste plaats is ontworpen voor moeilijk rekenkundig werk, ontbreken functie-procedures voor dit doel.
  Bijvoorbeeld:

```
(AANTAL * BEDRAG + 50)/100
```

- logische expressie:
  Deze wordt hier 'condition' genoemd en maakt gebruik van de relationele en logische operatoren.
  Bijvoorbeeld:

```
AANTAL > 1000 AND BEDRAG NOT = 100
CODE = 73 OR CODE = 75 OR CODE = 79
```

Voor de evaluatie van een expressie is het nodig de prioriteiten van de gebruikte operatoren te kennen. De prioriteiten voor de aritmetische operatoren in COBOL zijn:

prioriteit	operator
3	**
2	* /
1	+ -

Als we de prioriteiten toekennen gebeurt de waardeberekening als volgt:
- Als alle operatoren gelijke prioriteit hebben worden ze van links naar rechts uitgevoerd.
- Indien operatoren met verschillende prioriteit voorkomen, worden eerst de operatoren met de hoogste prioriteit uitgevoerd, vervolgens die met de op één na hoogste prioriteit, enzovoorts.
- Door middel van ronde haakjes zijn beide voorgaande regels te omzeilen: de waarde van de expressie binnen de haakjes wordt eerst berekend (volgens deze regels!).

Voor de evaluatie van logische expressies geldt:
- Eerst worden de aritmetische expressies geëvalueerd.

- Vervolgens worden de relationele operatoren uitgevoerd.
- Vervolgens de logische operatoren, waarbij NOT een hogere prioriteit heeft dan AND en deze op zijn beurt een hogere prioriteit dan OR.

Een vorm van de assignment statement in COBOL, die gebruik maakt van een aritmetische expressie is:

```
COMPUTE identifier-1 = < arithmetic expression >
```

Bijvoorbeeld:

```
COMPUTE B = 1.
COMPUTE B = A.
COMPUTE P = Q - R + S.
COMPUTE SOM = SOM + TERM/AANTAL.
COMPUTE K = L**N.
```

Ook bestaat er een aparte statement voor het overdragen van de waarde van een constante of van een variabele aan één of meer (andere) variabelen. De definitie van deze statement luidt:

```
     { literal      }
MOVE {              }  TO identifier-2 [ ,identifier-3]...
     { identifier-1 }
```

Bijvoorbeeld:

```
MOVE X TO Y.
MOVE X TO Y,Z.
```

Of de MOVE-statement is toegestaan en wat het effect ervan is, is afhankelijk van de beschrijving van de variabelen in de DATA DIVISION (lengte en type van PICTURE); in grote lijnen geldt dat, bij gelijke beschrijvingen, het effect van de laatste statement uit het voorgaande voorbeeld is dat Y en Z de waarde van X krijgen (bij ongelijke PICTURES treedt conversie op).

Naast de COMPUTE- en MOVE-vorm kent COBOL nog een aantal alternatieve notaties voor het berekenen en toekennen van waarden*.

```
    {identifier-1}  [,identifier-2]
ADD {            }  [              ] ... TO identifier-k [,identifier-n] ...
    {literal-1   }  [,literal-2    ]

    {identifier-1}  {,identifier-2}
ADD {            }  {             } ... GIVING identifier-k
    {literal-1   }  {,literal-2   }
```

* Om druktechnische redenen zijn deze notaties verkleind afgedrukt.

```
SUBTRACT {identifier-1} [,identifier-2] ... FROM identifier-m [,identifier-n] ...
         {literal-1   } [,literal-2   ]

SUBTRACT {identifier-1} [,identifier-2]...FROM {identifier-n} GIVING identifier-k
         {literal-1   } [,literal-2   ]        {literal-n   }

MULTIPLY {identifier-1} BY identifier-2
         {literal-1   }

MULTIPLY {identifier-1} BY {identifier-2} GIVING identifier-3
         {literal-1   }    {literal-2   }

DIVIDE   {identifier-1} INTO identifier-2
         {literal-1   }

DIVIDE   {identifier-1} {INTO} {identifier-2} GIVING identifier-3
         {literal-1   } { BY } {literal-2   }
```

VOORBEELDEN

COBOL:	Pascal equivalent:
ADD 100 TO SOM.	SOM := SOM + 100
ADD A, B, C TO D, E.	D := D + A + B + C; E := E + A + B + C
ADD X, Y, Z GIVING T.	T := X + Y + Z
SUBTRACT E, F FROM G, H.	G := G - E - F; H := H - E - F
SUBTRACT R, S FROM T GIVING U.	U := T - R - S
MULTIPLY A BY B.	B := A * B
MULTIPLY H BY 3.	fout
MULTIPLY X BY Y GIVING Z.	Z := Y * X
DIVIDE A BY B.	B := A/B.
DIVIDE A INTO B.	B := B/A.

8.6 REPETITIE

Pascal

Pascal biedt drie vormen van een repetitie:

<repetitive statement> ::= <while statement>
|<repeat statement>
|<for statement>

<while statement> ::= while <boolean expression> do <statement>

<repeat statement> ::= repeat <statement sequence> until <boolean expression>

```
<for statement>          ::= for <identifier> := <init> to <final>
                             do <statement>
                             |for <identifier> := <init>
                                  downto <final>
                             do <statement>
```

Hier staan <init> en <final> voor expressies van hetzelfde enkelvoudige type als de identifier (niet: real). Een <statement sequence> is een rij statements gescheiden door puntkomma's; een <statement> kan ook een <compound statement> zijn (zie § 8.2). Het effect van elk van deze drie statements is zoals in § 7.3 besproken is.

Bij de <for statement> doorloopt de <identifier> alle waarden in het interval [<init>..<final>] precies één keer; in het eerste geval 'van laag tot hoog', in het tweede geval 'van hoog tot laag'. De <for statement> heeft geen enkel effect als het interval leeg is. De waarde van de <identifier> is na afloop ongedefinieerd. Aan de <identifier> mag in de <statement> geen waarde worden toegekend.

VOORBEELD

Als het gemiddelde van 10 waarden in een array getal [1..10] berekend moet worden, kan het stukje programma hiervoor luiden:

```
som := 0; i := 0;
while i < 10 do begin i := i + 1;
                      som := som + getal[i]
                end;
gemiddelde := som/10;
```

Of:

```
som := 0; i := 0;
repeat i := i + 1; som := som + getal[i] until i = 10;
gemiddelde := som/10;
```

Of:

```
som := 0;
for i := 1 to 10 do som := som + getal[i];
gemiddelde := som/10;
```

Of:

```
som := 0;
for i := 10 downto 1 do som := som + getal[i];
gemiddelde := som/10;
```

(einde voorbeeld)

De <for statement> kan gebruikt worden als het aantal malen dat de <statement> herhaald moet worden bij voorbaat vast staat. Bij de <repeat statement> wordt de <statement sequence> ten minste één maal uitgevoerd!

FORTRAN

De repetitie heeft in FORTRAN de vorm:

```
DO <label> <variabele> = <icv> , <icv> , <icv>
```

waarin <icv> staat voor een integer constante of een enkelvoudige integer variabele. Het effect is dat alle statements, vanaf deze en tot en met die waar het label (= geheel getal van maximaal vijf cijfers) voor staat, worden herhaald. De <icv>'s geven respectievelijk de beginwaarde van de (integer) variabele, de eindwaarde van de variabele en de waarde waarmee de waarde van de variabele steeds na elke slag van de repetitie wordt opgehoogd (als dit 1 moet zijn, mag de derde <icv> met de er voor staande komma achterwege blijven). De twee eerste <icv>'s moeten groter dan 0 zijn en tijdens de uitvoering van de statement mogen deze en de lopende variabele zelf niet van waarde worden veranderd (de lopende variabele wordt door de DO statement aangepast). De derde <icv> mag ook negatief zijn. De repetitie wordt beëindigd, zodra de lopende variabele groter (bij positieve stapgrootte) of kleiner (bij negatieve stapgrootte) is geworden dan de eindwaarde. Vaak wordt als laatste statement van de rij de CONTINUE statement gebruikt, omdat er een aantal statements zijn waarmee de rij te herhalen statements niet mag eindigen (zoals de statements die beginnen met IF, GOTO en DO). Het voorbeeld voor het berekenen van het gemiddelde:

```
      SOM = 0.
      DO 200 K = 1,10
         SOM = SOM + GETAL(K)
  200 CONTINUE
      GEM = SOM/10.
```

COBOL

In COBOL zijn een aantal mogelijkheden voor repetitie. De statements die een aantal malen moeten worden uitgevoerd worden aangeroepen als een routine, doordat de naam van de paragraaf of de secties waarin deze statements staan wordt gebruikt. Hier bespreken we een drietal vormen.

VORM 1

```
PERFORM <procedure-name-1> [ THRU <procedure-name-2> ]
                           [ <int> TIMES ]
```

Hier staat <int> voor een integer variabele of constante.

De procedurenamen zijn namen van secties of paragrafen. Zowel een sectie als een paragraaf bestaat uit een aantal statements voorafgegaan door een naam; in § 8.9 komen we hierop terug.

Het gedeelte met TIMES geeft het aantal malen dat de statements worden uitgevoerd; als dit gedeelte ontbreekt, worden de statements éénmaal uitgevoerd. Als <int> een variabele is, mag deze in

de statements van de sectie of paragraaf van waarde worden veranderd; dit heeft geen invloed op het aantal malen dat de statements worden uitgevoerd.

Als de THRU-optie ontbreekt, worden de statements onder 'procedure-name-1' uitgevoerd; is de optie wel aanwezig, dan worden de statements onder 'procedure-name-1' tot en met de statements onder 'procedure-name-2' uitgevoerd.

Nadat de statements (het aantal malen) zijn uitgevoerd, wordt vervolgd met de statement volgend op de PERFORM statement.

VORM 2

PERFORM <procedure-name-1> [THRU <procedure-name-2>]
UNTIL <condition>

De <condition> is een logische expressie en deze bepaalt nu wanneer de repetitie eindigt. De statements worden uitgevoerd totdat de logische expressie de waarde true oplevert; is de waarde reeds aan het begin true dan worden de statements niet uitgevoerd (vergelijk dit met de in hoofdstuk 7 genoemde repeat statement en while statement).

Verder gelden dezelfde opmerkingen als hierboven.

Met deze vorm zou het voorbeeldje van het gemiddelde van een tabel GETAL van 10 elementen kunnen luiden:

```
   MOVE 0 TO SOM.
   MOVE 1 TO I.
   PERFORM S UNTIL I > 10.
   COMPUTE GEM = SOM/10.
   ....................
   ....................
   ....................
S. ADD GETAL(I) TO SOM.
   ADD 1 TO I.
```

VORM 3

PERFORM <procedure-name-1> [THRU <procedure-name-2>]
VARYING <identifier> FROM <nv> BY <nv>
UNTIL <condition>

Hier staat <nv> voor een numerieke variabele of constante.

Deze vorm kan nog worden uitgebreid; we laten dat hier achterwege.

De THRU-optie wordt hetzelfde behandeld als in de andere gevallen.

Deze vorm is de herhalingsopdracht zoals we die ook in de andere talen hebben gezien, met een lopende variabele (<identifier>), een beginwaarde (na FROM) en een waarde waarmee wordt opgehoogd (na BY). De enige afwijking is de logische expressie na UNTIL; deze logische expressie behoeft geen test op de lopende variabele te zijn, maar is dit in de regel natuurlijk wel. De statements worden uitgevoerd totdat de logische expressie de waarde

true oplevert. Is dit reeds in het begin het geval, dan worden de statements geen enkele keer uitgevoerd.

Met deze laatste vorm zou ons voorbeeld van het gemiddelde kunnen luiden:

```
   MOVE ZERO TO SOM.
   PERFORM S VARYING I
             FROM 1
               BY 1
            UNTIL I > 10.
   COMPUTE GEM = SOM/10.
   ....................
   ....................
S. ADD GETAL(I) TO SOM.
```

8.7 VOORWAARDELIJKE CONSTRUCTIES

Pascal

In Pascal zijn twee mogelijkheden om te kiezen welke statements uitgevoerd moeten worden, afhankelijk van een conditie.

```
<conditional statement>   ::= <if statement> | <case statement>
<if statement>            ::= if <boolean expression> then
                              <statement> [else <statement>]
<case statement>          ::= case <expression> of
                              <case l.s> {;<case l.s.>} end
<case l.s.>               ::= <case label list> : <statement>
                              |<empty>
<case label list>         ::= <constante> {, <constante>}
```

Een <statement> kan een <compound statement> zijn, zie § 8.2. De <expression> bij de <case statement> moet een waarde van een enkelvoudig type zijn anders dan real; de constanten in de <case label list> moeten waarden van hetzelfde type zijn als de <expression>.

Het effect van de <if statement> en de <case statement> is zoals in § 7.3 beschreven is.

VOORBEELDEN

- Het bepalen van de grootste gemene deler van de waarden van twee positieve integer variabelen a en b:

```
while a <> b do if a < b then a := a - b else b := b - a;
ggd := a;
```

- Het teken bepalen van een variabele x:

```
if x > 0 then sign := 1 else if x < 0 then sign := -1
                                        else sign := 0;
```

- Het bepalen van het aantal dagen in een maand in een bepaald jaar:

```
var maand: 1..12;
    jaar: 1900..2000;
maand := ...; jaar := ...;
case maand of
1, 3, 5, 7, 8, 10, 12: aantal := 31;
4, 6, 9, 11: aantal := 30;
2: if (jaar mod 4 = 0) and (jaar <> 1900)
   then aantal := 29
   else aantal := 28
end;
```

FORTRAN

De voorwaardelijke opdracht in FORTRAN heet de logical IF statement en heeft de vorm:

```
<conditional statement> ::= IF (<logical expression>)
                                <statement>
```

De <statement> mag niet beginnen met DO, IF, ELSE of END. Bijvoorbeeld:

```
IF (A.LT.B) C = D + 1
IF (A.GE.B) C = D
```

Daarnaast kent FORTRAN nog een constructiemogelijkheid (de IF-THEN statement), waarmee zowel voorwaardelijke als alternatieve constructies gerealiseerd kunnen worden:

```
<IF-THEN statement> ::= IF (<logical expression>)
                        THEN <statement list>
                        [ELSE <statement list>]
                        END IF
```

De <statement list> kan bestaan uit bijvoorbeeld assignment statements, repetities of weer de constructie zelf. In dit laatste geval (geneste IF-THEN) komt in de totale constructie echter slechts één END IF voor.
Bijvoorbeeld:

```
- IF (I.EQ.0) THEN
     PRINT*,J
  ELSE
     J = J/I
     PRINT*,I,J
  END IF
```

```
- IF (A.EQ.1) THEN
     C = C + 1
  ELSE IF (A.EQ.2) THEN
     E = E + 1
  ELSE IF (A.EQ.3) THEN
     G = G + 1
  ELSE
     I = I + 1
   END IF
```

Iedere regel in FORTRAN is eigenlijk een statement.

De alternatieve opdracht in FORTRAN kan dus gevormd worden met de volgende statements:
- IF (<logical expression>) THEN
- ELSE IF (<logical expression>) THEN
- ELSE
- END IF

Afhankelijk van de waarde van de logische expressie worden alle statements tussen THEN respectievelijk ELSE en de bijbehorende ELSE of END IF uitgevoerd.

Naast de hier beschreven opdrachten kent FORTRAN nog de voorwaardelijke sprongopdracht; wij zullen deze behandelen in de volgende paragraaf.

COBOL

In COBOL zijn een groot aantal voorwaardelijke constructies aanwezig; een aantal hiervan zijn uitbreidingen van statements die al behandeld zijn. We zullen hier alleen de IF statement behandelen:

$$\underline{\text{IF}}\ \text{CONDITION} \begin{Bmatrix} \text{sentence-1} \\ \underline{\text{NEXT}}\ \underline{\text{SENTENCE}} \end{Bmatrix} \begin{bmatrix} ;\ \underline{\text{ELSE}}\ \text{sentence-2} \\ ;\ \underline{\text{ELSE}}\ \underline{\text{NEXT}}\ \underline{\text{SENTENCE}} \end{bmatrix}$$

Een sentence bestaat uit één of meer statements. De statements mogen willekeurige statements zijn. De puntkomma's behoeven niet geschreven te worden. Het gedeelte "ELSE NEXT SENTENCE" mag ontbreken en men heeft dan een statement met dezelfde vorm en hetzelfde effect als de conditional statement uit de Inleiding; ook als dit gedeelte er wel staat heeft men hetzelfde effect als de conditional statement. De statements hebben verder hetzelfde effect als de conditie en selectie statements, waarbij nog opgemerkt moet worden dat

```
IF I NOT > 10 NEXT SENTENCE
ELSE COMPUTE GEM = SOM/10
```

hetzelfde effect heeft als

```
IF I > 10 COMPUTE GEM = SOM/10.
```

De conditional statement is steeds te interpreteren, doordat een ELSE steeds behoort bij het daaraan voorafgaande IF (dat nog niet aan een ELSE is gekoppeld). Als men bijvoorbeeld in een statement zou willen vastleggen:

S = -1 als X < 0
S = 0 als X = 0
S = 1 als X > 0

dan kan dit met:

```
IF X < 0 MOVE -1 TO S
ELSE IF X = 0 MOVE 0 TO S
     ELSE MOVE 1 TO S.
```

Als men de grootste waarde wil vinden van de drie variabelen A, B en C, dan kan dat met:

```
IF A > B IF A > C MOVE A TO MAX
         ELSE MOVE C TO MAX
   ELSE IF B > C MOVE B TO MAX
        ELSE MOVE C TO MAX.
```

Dit kan men ook (en veel overzichtelijker) schrijven als:

```
MOVE A TO MAX.
IF B > MAX MOVE B TO MAX.
IF C > MAX MOVE C TO MAX.
```

8.8 GOTO-STATEMENTS

Goto-statements kunnen onderscheiden worden in voorwaardelijke en onvoorwaardelijke. Als in een taal de voorwaardelijke goto-statement niet aanwezig is, kan deze gerealiseerd worden door combinatie van een voorwaardelijke constructie en een onvoorwaardelijke goto-statement.

Pascal

Een statement kan in Pascal voorafgegaan worden door een label, dat is een integer constante zonder teken. Deze labels moeten gedeclareerd worden! De goto-statement heeft de vorm:

goto <label>

en heeft als effect dat vervolgd wordt met de statement die volgt op die label. Als een label voor een statement binnen een repetitie, voorwaardelijke constructie of blok staat, mag buiten deze constructie niet een goto-statement met dat label voorkomen.

VOORBEELD
fout is:

```
repeat S1;
       10: S2;
        S3
until  a < b;
.....
goto 10;
```

goed is:

```
begin S1;
      if x < 0 then goto 1;
      S2
end;
.....
1: S3;
```

(einde voorbeeld)

Omdat in Pascal goede besturingselementen aanwezig zijn, is het gebruik van de goto-statement zelden nodig.

FORTRAN

Ook in FORTRAN bestaat de GO TO statement:

GO TO <label>

De <label> is in FORTRAN een geheel getal van ten hoogste vijf cijfers.

Een voorwaardelijke GO TO statement kan gerealiseerd worden door voor de statement in 'IF (<logical expression>) <statement>' een GO TO statement in te vullen, bijvoorbeeld:

```
IF (A.LT.B) GO TO 100
```

In FORTRAN zijn nog een drietal andere constructies aanwezig die als voorwaardelijke GO TO statements te beschouwen zijn, omdat de uitvoering van de GO TO statement afhankelijk is van een voorwaarde.

a. Assigned GO TO statement
Deze heeft de vorm:

<assigned GO TO statement> ::= GO TO <integer variable>,
(<list of labels>)
<list of labels> ::= <label> | <list of labels>, <label>

Op het moment dat deze statement wordt uitgevoerd moet de (integer) variabele de waarde van een van de labels hebben en er wordt dan vervolgd met de statement volgend op deze label.

b. Computed GO TO statement
Deze heeft de vorm:

<computed GO TO statement> ::= GO TO (<list of labels>),
<integer variable>

De labels in de lijst worden genummerd beschouwd van 1 tot en met n (als er n labels zijn in de lijst).

Op het moment dat de statement wordt uitgevoerd moet de variabele een waarde i hebben, waarvoor geldt $1 \leqq i \leqq n$, en er wordt vervolgd met de statement volgend op de i-de label uit de lijst. Als $i > n$ wordt vervolgd met de statement die volgt op de computed GO TO. Met deze opdracht kan de case statement uit hoofdstuk 7 worden gesimuleerd.

c. Arithmetic IF statement
Deze heeft de vorm:

<arithmetic IF statement> ::= IF (<arithmetic expression>)
<label>, <label>, <label>

Er wordt vervolgd met de statement volgend op de eerste label als de waarde van de expressie negatief is, met de statement volgend op de tweede label als de waarde nul is, en met de statement volgend op de derde label als de waarde positief is.

We kunnen het voorbeeld van het berekenen van het gemiddelde nu schrijven als:

```
    SOM = 0.
    K = 1
100 SOM = SOM + GETAL(K)
    K = K + 1
    IF (K.LE.10) GO TO 100
    GEM = SOM/10.
```

Hierbij hebben we gebruik gemaakt van de logical IF statement; bij gebruik van de arithmetic IF statement wordt het:

```
    SOM = 0.
    K = 1
100 SOM = SOM + GETAL(K)
    K = K + 1
    IF (K-10) 100, 100, 200
200 GEM = SOM/10.
```

COBOL

De GO TO statement heeft in COBOL de vorm:

GO TO <procedure-name>

Hierin is de <procedure-name> de naam van een paragraaf of van een sectie (zie § 8.2), en er wordt gesprongen naar de eerste

opdracht van de genoemde paragraaf of sectie.
In ons voorbeeld:

```
      MOVE ZERO TO SOM.
      MOVE 1 TO I.
   A. ADD GETAL(I) TO SOM.
      ADD 1 TO I.
      IF I NOT > 10 GO TO A.
      COMPUTE GEM = SOM/10.
```

In COBOL is ook een tweede vorm van de GO TO statement aanwezig:

GO TO <procedure-name-1>, <procedure-name-2>, ...,
<procedure-name-n> DEPENDING <identifier>

De identifier moet een gehele waarde hebben. Het aantal procedurenamen moet groter dan of gelijk aan 1 zijn. Als er n procedurenamen zijn worden deze genummerd beschouwd van 1 tot en met n.

Als de waarde van de identifier i is ($1 \leqq i \leqq n$), wordt vervolgd met de sectie of paragraaf met de i-de naam uit de lijst; als i < 1 of i > n is wordt vervolgd met de eerstvolgende statement.

8.9 PROCEDURES EN FUNCTIES

Pascal

Een procedure of function declaration heeft in Pascal de vorm:

function <identifier> [<formal parameter list>] :
<type identifier>; <block>
procedure <identifier> [<formal parameter list>] ; <block>

met:

```
<formal parameter list > ::= (<formal par>{; <formal par>})
<formal par> ::= <parameter group>
               | var <parameter group>
               | procedure <identifier list>
               | function <parameter group>
<parameter group> ::= <identifier list> : <type-identifier>
```

De formele parameters worden verdeeld in vier klassen:
- value parameters, die 'by value' worden overgedragen;
- variabele (var) parameters, die 'by reference' worden overgedragen;
- procedure parameters;
- function parameters.

De aanroep van een procedure of function gaat met:

<identifier> [<actual parameters list>]

De procedure-aanroep is een statement; de function-aanroep gebeurt in een expressie:

```
<actual parameter list> ::= (<actual par> {; <actual par>})
<actual par> ::= <expression>
               | <variable>
               | <procedure identifier>
               | <function identifier>
```

Het aantal parameters in de formele en de actuele lijst moet natuurlijk hetzelfde zijn. Elke actuele parameter correspondeert met de formele die op dezelfde plaats in de lijst staat; het type en de klasse (value, variabele, procedure of function) van de actuele en de formele moeten overeenkomen.

Alle actuele variabele parameters voor een procedure of function-aanroep moeten verschillend zijn.

In het <block> van de function declaratie moet ten minste één waardetoekenning (van het gespecificeerde type) aan de <function identifier> gebeuren. In het <block> van een procedure (of function) mag de procedure (function) zelf aangeroepen worden; recursie is dus mogelijk. Indien verschillende procedures of functies elkaar wederzijds aanroepen, spreken we van indirecte recursie. Bij de declaratie zal in het <block> van de eerst gedeclareerde dan de naam van een andere, nog niet gedeclareerde procedure of function voorkomen. Om te vermijden dat in het programma een nog niet gedeclareerde naam voorkomt moet in zo'n geval een *forward declaratie* plaatsvinden, waarbij het <block> van deze procedure/function vervangen is door het reserved word forward. Later moet dan de echte declaratie volgen, waarbij de eventuele <formal parameter list> en het type niet herhaald moeten worden.

VOORBEELD

```
function B (x: integer): char; forward;
procedure A (var y: char);
begin .....
      y := B(i);
      .....
end;
function B;
begin .....
      A(c);
      .....
end;
```

(einde voorbeeld)

In § 8.5 zagen we reeds dat er in Pascal een groot aantal standaardfuncties zijn. Er zijn ook een aantal standaardprocedures, onder andere voor file-operaties en pointer-operaties.

FORTRAN

Een subroutine in FORTRAN kan worden opgevat als een zelfstandig programma dat vanuit het hoofdprogramma kan worden aangeroepen door een CALL statement. De statement CALL PIET in het hoofdprogramma roept de subroutine aan met de naam PIET. De subroutine zelf begint met het symbool SUBROUTINE, dan de naam, hier PIET, en dan de formele parameters tussen haakjes. De subroutinetekst eindigt met het symbool END. (Opmerking: in het voorbeeld hierboven heeft PIET geen parameters.)

De parameters worden gebruikt in de zin van 'call by reference'. In de subroutine wordt met een RETURN statement het dynamisch einde aangegeven; er mogen verscheidene RETURN-statements voorkomen.

De variabelen in een subroutine zijn lokaal, dus alleen in de routine bekend. Waarden worden van hoofdprogramma naar subroutine of van subroutine naar subroutine overgedragen via het parametermechanisme. Subroutines mogen in FORTRAN subroutines aanroepen, echter niet recursief.

In § 8.5 bespraken we reeds een aantal standaardfuncties die in aritmetische expressies kunnen worden gebruikt. Als de programmeur dat wil, kan hij zelf ook functies introduceren; deze moeten in één aritmetische waardetoekenningsopdracht kunnen worden geschreven en worden 'arithmetic statement functions' genoemd. Ze moeten komen vóór de eerste executabele statement in het programma en de naam moet eindigen met een F.

Bijvoorbeeld: ABF(A, B) = A ** 2 - B ** 2

De functie kan dan later in een aritmetische expressie opgeroepen worden, bijvoorbeeld in Z = X + Y + ABF(X, Y).

Een grote beperking van de arithmetic statement function is, dat de functie moet kunnen worden geschreven in één expressie. In FORTRAN kennen we ook nog het FUNCTION subprogramma dat veel op een SUBROUTINE lijkt. De FUNCTION begint met het woord FUNCTION, dan volgt de naam met eventuele formele parameters. De FUNCTION eindigt, net als de SUBROUTINE, statisch met END, en dynamisch met RETURN.

Voor de namen gelden dezelfde regels als bij enkelvoudige variabelen: beginnend met I tot en met N betekent INTEGER, anders REAL (men kan dit doorbreken door voor het woord FUNCTION het type op te geven, bijvoorbeeld REAL FUNCTION L).

VOORBEELD

```
   FUNCTION GEM(A, N)
      DIMENSION A(100)
      SOM = 0.
      DO 1 J = 1, N
   1  SOM = SOM + A(J)
      B = FLOATF(N)
      GEM = SOM/B
      RETURN
      END
```
(einde voorbeeld)

Een FUNCTION wordt aangeroepen door het gebruik van de naam, met eventuele actuele parameters, in een expressie.
Een SUBROUTINE wordt aangeroepen door een CALL statement.

We zagen reeds dat, afgezien van de parameters, alle variabelen in FORTRAN in een SUBROUTINE en in een FUNCTION lokaal zijn (dus niet in het hoofdprogramma bekend zijn). Als we variabelen in een aantal delen van het programma willen gebruiken, kan dit met de COMMON statement. Als we de variabelen A, B en C in meer programmadelen (hoofdprogramma en subprogramma's) dezelfde grootheden willen laten voorstellen, schrijven we in ieder van die delen:

COMMON A, B, C

of we schrijven bijvoorbeeld in het hoofdprogramma:

COMMON A, B, C

en in het subprogramma:

COMMON U, V, W

waarmee A en U dezelfde identiteit hebben (op de naam na), evenzo B en V en C en W. Hiermee kunnen globale variabelen worden geïntroduceerd, terwijl de programmadelen toch door verschillende programmeurs kunnen worden gemaakt.

Als men verschillende zogenaamde COMMON-gebieden wil gebruiken volstaat het genoemde mechanisme niet, omdat er geen onderscheid gemaakt kan worden tussen de verschillende gebieden. De gebieden moeten in dat geval een naam krijgen, bijvoorbeeld:

COMMON / NAAM / A, B

Het gebied heeft nu de naam NAAM gekregen.

VOORBEELD
Stel dat in een hoofdprogramma is opgenomen

COMMON PI

en dat de variabele PI in het hoofdprogramma een waarde (3.14...) heeft gekregen. We zouden dan als functies kunnen opnemen:

```
REAL FUNCTION BOL(STRAAL)
   REAL STRAAL
   REAL PI
   COMMON PI
BOL = (4.0/3.0) * PI * STRAAL ** 3
RETURN
END

REAL FUNCTION CYL(STRAAL, HOOGTE)
   REAL STRAAL, HOOGTE
   REAL PI
   COMMON PI
CYL = HOOGTE * PI * STRAAL ** 2
RETURN
END

REAL FUNCTION KEGEL(STRAAL, HOOGTE)
   REAL STRAAL, HOOGTE
   REAL PI
   COMMON PI
KEGEL = HOOGTE * PI * STRAAL ** 2/3.0
RETURN
END
```

(Het volume van een bol met straal r is $4\pi r^3/3$, het volume van een cilinder met straal r en hoogte h is $\pi r^2 h$ en het volume van een kegel met straal r van het grondoppervlak en hoogte h is $\pi r^2 h/3$.)
(einde voorbeeld)

We kunnen ook door de EQUIVALENCE statement verschillende variabelen eenzelfde identiteit geven. Bijvoorbeeld:
EQUIVALENCE(A, B)
Overal waar nu een A een bepaalde waarde krijgt, krijgt B ook die waarde.

COBOL

Zoals we al hebben gezien kan een gedeelte van een programma worden aangeroepen door een PERFORM statement. Bijvoorbeeld:

```
PERFORM BEREKENING.
```

Ergens in het programma moet dan een gedeelte (een paragraaf of sectie) de naam BEREKENING hebben en dat gedeelte wordt uitgevoerd. Als dit gedeelte is verwerkt, wordt vervolgd met de statement die volgt op de PERFORM.

Recursie is niet toegestaan, en met parameters en lokale variabelen kan niet gewerkt worden.

Met behulp van de CALL statement kan een subprogramma mèt

parameters aangeroepen worden. Het subprogramma zelf is een compleet programma met
- in de IDENTIFICATION DIVISION de naam (bijvoorbeeld FACT);
- in de DATA DIVISION een 'Linkage Section' waarin de structuur van de parameters gespecificeerd wordt;
- de namen van de parameters in de header van de PROCEDURE DIVISION (bijvoorbeeld PROCEDURE DIVISION USING N, F.);
- en één of meer EXIT statements in de PROCEDURE DIVISION.

De aanroep van zo'n subprogramma geschiedt met de CALL statement, waarbij de actuele parameters genoemd worden. Bijvoorbeeld:

```
CALL FACT USING NN, NFACT.
```

De parameters worden 'by reference' overgedragen. Recursie is niet toegestaan.

VOORBEELD

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. FACT.
ENVIRONMENT DIVISION.
 .....
DATA DIVISION.
 .....
 LINKAGE SECTION.
   77 N PIC S9(9) COMP.
   77 F PIC S9(9) COMP.
PROCEDURE DIVISION USING N, F.
   MOVE 1 TO F.
   PERFORM S1 VARYING I FROM 1 BY 1 UNTIL I = N OR I > N.
      S1.COMPUTE F = F * I.
      S2.EXIT PROGRAM.
IDENTIFICATION DIVISION.
 .....
 .....
PROCEDURE DIVISION.
 .....
 CALL FACT USING NN, NFACT.
 .....
 STOP RUN.
```

Een andere mogelijkheid is de ENTER statement; deze biedt de mogelijkheid programma's, geschreven in een andere taal dan COBOL, aan te roepen.

8.10 INVOER EN UITVOER

Pascal

In Pascal kan een programma werken met een invoer file en een uitvoer file, die beide sequentieel verwerkt worden. Het medium waarop deze files gerepresenteerd worden is voor het programma niet relevant. Beide files zijn gestructureerd als karakterregels. De namen van deze files worden in de program <heading> gespecificeerd. Bijvoorbeeld:

```
program P(input, output);
```

Binnen het programma werken de in- en uitvoeroperaties op de gespecificeerde files; de naam van de desbetreffende file hoeft niet aangeduid te worden.

Invoeroperaties

```
read(input, <variable list>);
```

of:

```
read(<variable list>);
```

Aan de variabelen in de <variable list>, die van het type integer, real, char of een subrange van integer of char moeten zijn, worden sequentieel de volgende waarden uit de file input toegekend.

VOORBEELD
Met de statement

```
read(input, rvar, cvar1, cvar2, ivar);
```

en de waarde van de invoer file input:

```
123.4AB56
```

wordt hetzelfde effect bereikt als met:

```
rvar := 123.4; cvar1 := 'A'; cvar2 := 'B'; ivar := 56;
```

(einde voorbeeld)

Alle waarden in de invoer file worden ten hoogste één keer gelezen. Voor aritmetische variabelen wordt de waarde genomen van de rij karakters die zo'n aritmetische waarde representeert, tot aan een spatie, een regelovergang of een karakter dat niet als onderdeel van het getal kan voorkomen. De typen moeten steeds overeenkomen.

Expliciete overgang naar een nieuwe regel in de invoer file wordt gerealiseerd met:

```
readln;
```

of:

```
readln(input);
```

Met:

readln(<variable list>);

of:

readln(input, <variable list>);

wordt overgang naar een nieuwe regel gerealiseerd nadat aan alle variabelen in de <variable list> een waarde is toegekend.

De test of het einde van een regel in de invoer file is bereikt kan worden uitgevoerd met de standaard boolean function:

eoln;

of:

eoln(input);

De test of het einde van de invoer file bereikt is met:

eof;

of:

eof(input);

Uitvoeroperaties

write(<output list>);

of:

write(output, <output list>);

Sequentieel worden de waarden (boolean of aritmetische expressies of strings, dat zijn rijtjes karakters) in de <output list> aan de uitvoer file output toegevoegd.

Voor boolean expressies worden de strings TRUE of FALSE toegevoegd. Voor de strings die string zelf. Voor real en integer expressies wordt een standaard aantal karakters toegevoegd, waarin de waarde gerepresenteerd staat. Dat aantal kan ook gespecificeerd worden bij de <output list>; we gaan daar verder niet op in.

Overgang naar een nieuwe regel wordt gerealiseerd met:

writeln;

of:

writeln(output);

Met:

writeln(<output list>);

of:

writeln(output, <output list>);

wordt de overgang naar een nieuwe regel gerealiseerd nadat alle waarden in de <output list> aan de uitvoer file zijn toegevoegd. De overgang naar een nieuwe pagina (bijvoorbeeld voor de regeldrukker) wordt gerealiseerd met:

page;

of:

page(output);

FORTRAN

FORTRAN biedt vele faciliteiten voor sequentieel lezen van, of schrijven naar invoer- en uitvoermedia (bijvoorbeeld: kaartlezer, regeldrukker, bestanden). We zullen er enkele van bespreken.

Standaardmedia
Deze zijn installatie-afhankelijk, meestal een kaartlezer en een regeldrukker. Per in- of uitvoerstatement worden een aantal waarden, tezamen een record vormend (bijvoorbeeld één ponskaart, één regel), gelezen of geschreven; elke statement verwerkt één of meer nieuwe records, nul of meer waarden. Hoe de karakters bij invoer geïnterpreteerd worden, of hoe de waarden bij uitvoer gerepresenteerd worden, kan worden aangegeven door middel van een FORMAT statement:

```
<nr> FORMAT(<format codes>)
```

waarbij <nr> een label is, en de <format codes> de lay-out van één of meer waarden beschrijven.

VOORBEELD

```
53 FORMAT(2X, I3, 10X, F5.2)
```

geeft aan: 2 karakters als spatie,
3 karakters als integer waarde,
10 karakters als spatie, en
5 karakters als real waarde interpreteren/representeren.
(einde voorbeeld)

Indien zonder FORMAT gewerkt wordt (aangegeven met: *), worden automatisch standaard formaten gehanteerd.

INVOER, ongeformatteerd:

```
READ *, <variable list>
```

De <variable list> is een rij van nul of meer variabelen of array elementen waaraan sequentieel een waarde wordt toegekend. De waarden moeten op het invoermedium gescheiden zijn door een komma of door één of meer spaties.

INVOER, geformatteerd:

```
READ <formatnr>, <variable list>
```

Het te gebruiken FORMAT geeft aan hoe de rij invoerkarakters als rij waarden moet worden geïnterpreteerd. Bijvoorbeeld:

```
READ 53, I, R
```

kent met bovenstaand FORMAT aan I en R de waarden toe die in het volgende record (op de volgende ponskaart) staan gerepresenteerd in karakters 3 tot en met 5 respectievelijk 16 tot en met 20.

UITVOER, ongeformatteerd:

PRINT *, <output list>

De <output list> is een rij van nul of meer rekenkundige expressies of tekstwaarden. De waarden worden gescheiden door een spatie.

UITVOER, geformatteerd:

PRINT <formatnr>, <output list>

Het te gebruiken FORMAT geeft aan hoe de rij waarden als rij karakters wordt weergegeven. Bijvoorbeeld:

PRINT 53, I, R

beschrijft het volgende record (de volgende regel) met: eerst twee spaties, dan drie karakters om de integer waarde van I te representeren, vervolgens 10 spaties en tot slot 5 karakters om de real waarde van R te representeren.

Bestanden
Per bestand worden de waarden als karakters opgeslagen in records die alle dezelfde lengte hebben en dezelfde indeling; de indeling is vastgelegd in een zogenaamd recordprofiel. Per in- of uitvoerstatement kan een record gelezen of een record beschreven worden (steeds: het volgende record). Om welk bestand het gaat wordt bij de in- of uitvoerstatement aangegeven door het bestandnummer; aldus is meteen het format van het record bekend.

INVOER:

READ(<bestandnummer>) <variable list>

Het recordprofiel en de typen van de in de lijst genoemde variabele(n) moeten met elkaar overeenkomen.

UITVOER:

WRITE(<bestandnummer>) <variable list>

De elementen van de lijst zijn array elementen, variabelen of volledige arrays.

Voor het correct werken met bestanden zijn verder de REWIND-statement en de ENDFILE-statement (afsluitrij) beschikbaar.

COBOL

In COBOL is de invoer gebaseerd op het werken met files en records. Bij uitstek karakteristiek voor COBOL zijn de uitgebreide mogelijkheden om met grote gegevensverzamelingen (bestanden, in een COBOL-programma alle aangeduid als FILE) te werken. De structuur en de mogelijkheden van de taal zijn geheel op dit gebruik afgestemd.

Er zijn twee mogelijke verwerkingswijzen van bestanden.

a. Sequentieel, hetgeen wil zeggen dat een bestand wordt verwerkt (gelezen of beschreven) door de records één voor één te verwerken.
Met andere woorden: een record kan slechts dan worden gelezen als alle voorgangers zijn verwerkt.

b. Direct (willekeurig, random), hetgeen wil zeggen dat records in een bestand worden verwerkt in een volgorde die wordt bepaald door het programma. Hierbij is het noodzakelijk dat de records in het bestand adresseerbaar zijn.

Er is geen bovengrens voor het aantal bestanden dat in één COBOL-programma verwerkt kan worden.

De opdrachten voor het lezen en schrijven van records van (in) een FILE zijn zeer eenvoudig:

lezen : READ file-name
schrijven : WRITE record-name

In de FILE SECTION is aangegeven welke recordstructuur bij welke file hoort.

Dit betekent evenwel niet dat de COBOL-programmeur niets te maken heeft met zaken als verwerkingswijze van de records, organisatievorm van het bestand, blokkingsfactor, record-lengte en dergelijke. Het vertaalprogramma heeft deze en andere gegevens nodig om de juiste doeltaalcode te genereren. Dergelijke fysieke gegevens worden gegeven op de plaatsen waar de bestanden en records worden gedeclareerd: in de paragraaf FILE-CONTROL van de ENVIRONMENT DIVISION en in de FILE SECTION van de DATA DIVISION. Voordeel hiervan is, dat tijdens het programmeren zèlf alleen logische grootheden een rol spelen, en bovendien dat, wanneer een fysieke grootheid verandert, de verandering slechts op één plaats in het programma behoeft te worden doorgevoerd.

Veel aandacht is besteed aan taalelementen die het afdrukken van lijsten en rapporten eenvoudiger kunnen maken. Zo zijn er uitgebreide mogelijkheden om de regelopschuiving te regelen (automatische bladopschuiving, automatisch regels overslaan, op dezelfde regel iets bijdrukken).

Ook aan de regelopmaak is veel aandacht besteed. Niet alleen is het mogelijk de regelindeling zelf vast te stellen, binnen één lijst of rapport kunnen ook nog verschillende soorten regels (met elk een eigen formaat) voorkomen.

Tenslotte is er nog de mogelijkheid de gegevens die moeten worden afgedrukt netjes 'op te maken' (Engels: editing). Zo is het bijvoorbeeld mogelijk op bepaalde posities vaste tekens toe of tussen te voegen, en ook niet-relevante nullen te onderdrukken. De specificaties voor deze opmaak van gegevens geschiedt in het PICTURE. Als bijvoorbeeld is gedeclareerd:

```
02 BEDRAG PICTURE ZZ,ZZ9.99.
```

hebben de volgende opdrachten het daarbij aangegeven effect met betrekking tot de inhoud van BEDRAG:

```
MOVE 70000 TO BEDRAG          70,000,00
MOVE 7000  TO BEDRAG           7,000.00
MOVE 700   TO BEDRAG             700.00
MOVE 70    TO BEDRAG              70.00
MOVE 7     TO BEDRAG               7.00
MOVE 0.7   TO BEDRAG               0.70
MOVE 0.07  TO BEDRAG               0.07
```

8.11 VOORBEELDEN

We behandelen in deze paragraaf voor elk van de drie talen een (min of meer) specifiek probleempje en geven daarbij het volledige programma. Een vergelijking van Pascal, FORTRAN en COBOL, aan de hand van de drie programma's, is wegens de verschillende aard van de problemen niet mogelijk. Deze paragraaf is slechts bedoeld ter illustratie van de eerder behandelde mogelijkheden en constructies van de drie talen.

Pascal

Het probleem heeft de naam 'De torens van Hanoi'. Gegeven zijn drie pinnen (die we 'links', 'midden' en 'rechts' zullen noemen). Op links zijn n schijven geplaatst met van beneden naar boven een steeds afnemende diameter. Gevraagd wordt de n schijven naar rechts te verplaatsen (met midden als hulppin) waarbij de volgende regels in acht moeten worden genomen:
- er mag steeds maar één schijf tegelijk verplaatst worden;
- een schijf mag nooit liggen op een schijf met een kleinere diameter;
- de n schijven liggen na iedere verplaatsing op een van de drie pinnen.

Als het probleem oplosbaar is voor n-1 schijven, is het oplosbaar voor n schijven. Want in dat geval verplaatsen we de n-1 bovenste schijven van links naar midden (met rechts als hulppin), dan verplaatsen we de onderste schijf van links naar rechts om daarna de n-1 schijven van midden naar rechts te verplaatsen (met links als hulppin). We kunnen ons hierbij aan de regels houden. Nu nog de oplossing voor n-1 schijven. Deze is te vinden als het probleem oplosbaar is voor n-2 schijven, etc. We komen tenslotte uit bij het geval van 1 schijf en dit is zeker oplosbaar. Deze aanpak leidt tot een recursieve oplossing.

```
program TowersofHanoi(input, output);
   type pin = (links, midden, rechts);
        aantal = 1..maxint;
   var n: aantal;
   procedure verplaats(k: aantal;
                       bron, hulp, doel: pin);
         procedure eenschijf;
               procedure printpin(p: pin);
                     begin case p of
                             links:  write('links');
                             midden: write('midden');
                             rechts: write('rechts')
                           end
                     end; {van printpin}
               begin write('verplaats een schijf van');
                     printpin(bron); write('naar');
                     printpin(doel); writeln
               end; {van eenschijf}
         begin if k = 1
               then eenschijf
               else begin verplaats(k-1), bron, doel, hulp);
                          eenschijf;
                          verplaats(k-1, hulp, bron, doel)
                    end
         end; {van verplaats}
    begin read(n);
          writeln('voor', n, 'schijven zijn de verplaatsingen:');
          writeln;
          verplaats(n, links, midden, rechts)
    end.
```

FORTRAN

De procedure EVENLIJST heeft twee parameters: de integer M en een array A van M integer waarden. Het effect van EVENLIJST is dat de even waarden uit A afgedrukt worden, en indien die niet aanwezig zijn wordt afgedrukt 'ALLE WAARDEN ONEVEN'. Hiertoe worden alle even waarden uit A in een array B opgeslagen en geteld in de teller L; afhankelijk van de waarde van L wordt hierna afgedrukt. Om te bepalen of een waarde even of oneven is wordt gebruik gemaakt van de functie EVEN die de waarde 1 aflevert als de parameter P even is, en anders 0. De procedure EVENLIJST wordt aangeroepen voor het array E dat met (ongeformatteerde) invoerstatements een waarde gekregen heeft.

```
      INTEGER E(100), I
      READ *, N
      DO 10 I = 1, N
       READ *, E(I)
   10 CONTINUE
      CALL EVENLIJST(N, E)
      END
      SUBROUTINE EVENLIJST(M, A)
      INTEGER A(100), M, K, B(100), I, EVEN
      L = 0
      DO 20 I = 1, M
       IF(EVEN(A(i)) .EQ. 1) THEN
                                   L = L + 1
                                   B(L) = A(L)
       END IF
   20 CONTINUE
      IF(L .EQ. 0) THEN
                         PRINT *, 'ALLE WAARDEN ONEVEN'
                    ELSE
                         PRINT *,L, (B(I), I = 1, L)
      END IF
      RETURN
      END
      INTEGER FUNCTION EVEN(P)
      INTEGER P, R
      IF(P .EQ. INT(P/2)*2) THEN
                                  R = 1
                             ELSE
                                  R = 0
      END IF
      EVEN = R
      RETURN
      END
```

COBOL

Het probleem luidt: Voor een aantal werknemers moeten toeslagen worden berekend. De toeslag is opgebouwd uit een basistoeslag, een leeftijdstoeslag (die x gulden bedraagt voor ieder jaar dat de werknemer ouder is dan 18 jaar) en een kindertoeslag (die y gulden bedraagt voor ieder kind). In de invoer file staan allereerst de bedragen x en y (als real waarde) en verder voor iedere werknemer zijn salarisnummer (integer), zijn basistoeslag (real), zijn leeftijd (integer) en het aantal kinderen (integer) dat hij heeft. Deze gegevens zijn gegroepeerd in groepen (records) van vier waarden, voor iedere werknemer een groep (record). Als uitvoer is gevraagd een rij records met voor iedere werknemer het salarisnummer, de basistoeslag, de leeftijd, het aantal kinderen en de uiteindelijke totale toeslag.

```
100000 IDENTIFICATION DIVISION.
100050 PROGRAM-ID. TOESLAGBEREKENING.
100100 AUTHOR. SCHRIJVER.
100150 DATE-WRITTEN. NU.
100250 ENVIRONMENT DIVISION.
100300 CONFIGURATION SECTION.
100350 SOURCE-COMPUTER.N1.
100400 OBJECT-COMPUTER.N2.
100450 INPUT-OUTPUT SECTION.
100500 FILE-CONTROL.
100550     SELECT INVOER ASSIGN TO KAARTLEZER.
100600     SELECT UITVOER ASSIGN TO REGELDRUKKER.
200000 DATA DIVISION.
200050 FILE SECTION.
200100 FD INVOER
200150     RECORD CONTAINS 80 CHARACTERS
200200     DATA RECORDS ARE KAART, KXY.
200250 01 KAART.
200300     02 SNR PICTURE 9999.
200350     02 BT  PICTURE 9(10).
200400     02 LFT PICTURE 99.
200450     02 AK  PICTURE 99.
200475     02 FILLER PICTURE X(62).
200500 01 KXY.
200550     02 RX PICTURE 9(6).
200600     02 RY PICTURE 9(5).
200625     02 FILLER PICTURE X(69)
200650 FD UITVOER
200700     RECORD CONTAINS 132 CHARACTERS
200750     DATA RECORD IS DRUKREGEL.
200800 01 DRUKREGEL.
200850     02 U1     PICTURE 9999.
200900     02 FILLER PICTURE X.
200950     02 U2     PICTURE 9(10).
201000     02 FILLER PICTURE X.
201150     02 U3     PICTURE 99.
201200     02 FILLER PICTURE X.
201250     02 U4     PICTURE 99.
201300     02 FILLER PICTURE X.
201350     02 U5     PICTURE 9(12).
201400     02 FILLER PICTURE X(98).
```

```
300000 WORKING-STORAGE SECTION.
300050 01 EIND.
300100    02 TEKST  PICTURE X(16) VALUE 'EINDE BEREKENING'
300150    02 FILLER PICTURE X(116).
300175 77 A         PICTURE 9.
300200 77 X         PICTURE 9(6).
300250 77 Y         PICTURE 9(5).
300300 77 FACTOR    PICTURE 99.
400000 PROCEDURE DIVISION.
400050 P1.
400100    OPEN INPUT INVOER.
400125    OPEN UITVOER.
400150    READ INVOER AT END MOVE -1 TO A.
400200    IF A NOT EQUAL -1  MOVE RX TO X
400250                       MOVE RY TO Y.
400300    READ INVOER AT END MOVE -1 TO A.
400350    PERFORM P2 UNTIL A = -1.
400400    WRITE DRUKREGEL FROM EIND AFTER 2 LINES.
400450    CLOSE INVOER, UITVOER.
400500    STOP RUN.
400550 P2.
400600    SUBTRACT 18 FROM LFT GIVING FACTOR.
400650    IF FACTOR LESS 0 THEN MOVE 0 TO FACTOR.
400700    MOVE SPACES TO DRUKREGEL.
400750    COMPUTE U5 = BT + FACTOR × X + AK × Y.
400800    MOVE SNR TO U1.
400850    MOVE BT TO U2.
400900    MOVE LFT TO U3.
400940    MOVE AK TO U4.
401000    WRITE DRUKREGEL.
401050    READ INVOER AT END MOVE -1 TO A.
```

9 SPECIALE PROGRAMMEERTALEN

9.1 INLEIDING

In de voorgaande hoofdstukken zijn algemene programmeertalen besproken; 'algemeen' in de zin van 'veel toegepast' en in de zin van 'geschikt voor een aantal toepassingsgebieden'. Dat deze talen algemeen zijn wil uiteraard niet zeggen dat er geen onderlinge verschillen zijn. Er zijn zeer veel algemene programmeertalen, er is slechts een klein aantal dat veel gebruikt wordt. Er zijn ook zeer veel speciale programmeertalen; 'speciaal' in de zin van 'bedoeld voor specifieke toepassingen' en in de zin van 'uitgerust met afwijkende constructiemogelijkheden'. Hier zullen een viertal speciale talen bekeken worden – LISP, SNOBOL, APL en SIMULA – en dan vooral om hun afwijkende constructies. De behandeling van de vier talen is er niet op gericht om een zodanige kennis te verschaffen dat men in deze talen zou kunnen programmeren. In de opgegeven literatuur vindt men uitvoeriger beschrijvingen van de talen.

9.2 LISP

LISP (LISt Processing) is door J. McCarthy omstreeks 1960 ontworpen voor het werken met lijsten van symbolische gegevens.

In het overzicht van de taal dat we hier geven, beperken we ons voornamelijk tot lijstverwerking en besteden geen aandacht aan meer 'klassieke' faciliteiten die ook in andere programmeertalen voorkomen. De met deze klassieke faciliteiten uitgebreide LISP-versie wordt LISP 1.5 genoemd.

LISP heeft drie eigenschappen waardoor de taal zich in het bijzonder onderscheidt van andere hogere programmeertalen:

1. De reeds genoemde afstemming in operaties en datastructuren op het verwerken van lijsten (lists; op te vatten als sequences).
2. Voor een groot deel wordt de volgorde van operaties bepaald door recursieve structuren.

3. Een LISP-programma heeft dezelfde opbouw als een LISP-datastructuur, waardoor met programma's gemanipuleerd kan worden als met gegevens en omgekeerd datastructuren opgevat kunnen worden als programma's.

Een LISP-programma bestaat uit een rij definities van functies, gevolgd door een rij aanroepen van deze functies. De functies worden gedefinieerd in de vorm van expressies. Iedere operator wordt ook geschreven als een functie. Er zijn enkele uitbreidingen in de taal opgenomen om deze functie-structuur te doorbreken maar de functie blijft de voornaamste bouwsteen.

De enige datastructuur in LISP is de symbolic-expression (meestal S-expressie genoemd), die òf enkelvoudig is (en een waarde wordt dan een atoom genoemd) òf samengesteld (en dan opgevat kan worden als een sequence van sequences). De besturingsstructuren in een LISP-programma zijn eigenlijk beperkt. Functies (alle operaties) worden geschreven in prefix-notatie. Er is een mogelijkheid voor het formuleren van een conditie en er kan gebruik worden gemaakt van een soort sprongopdracht. Verder wordt er erg veel gebruik gemaakt van recursie.

Parameters worden in de meeste gevallen overgedragen 'by value', maar ook 'by name' komt voor.

9.2.1 S-expressies

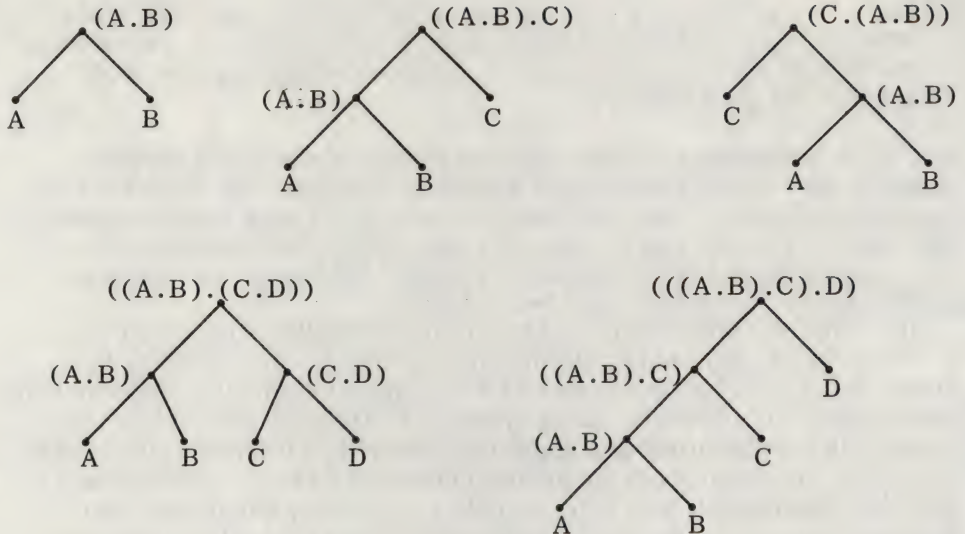
Een S-expressie is atomair of samengesteld. Een atomaire S-expressie wordt genoteerd als een identifier en kan willekeurige informatie (bijvoorbeeld een getalwaarde of een functie) voorstellen. Voor een niet-atomaire S-expressie bestaan twee (syntactisch verschillende) notaties.

We zullen ons hier in eerste instantie beperken tot de zogenaamde punt-notatie. In deze notatie wordt een niet-atomaire S-expressie geschreven als een paar van S-expressies. Als a en b S-expressies zijn (al dan niet atomair), dan is (a.b) een S-expressie.

Stel dat we atomen voorstellen door hoofdletters, dan zijn de volgende rijtjes S-expressies:

```
A
(A.B)
((A.B).C)
(C.(A.B))
((A.B).(C.D))
(((A.B).C).D)
```

Niet-atomaire S-expressies kunnen voorgesteld worden door binaire bomen, waarbij de atomaire componenten in de bladeren voorkomen. Voor de bovenstaande niet-atomaire expressies zijn de binaire bomen:



De andere notatie voor S-expressie is de zogenaamde list-notatie. Een niet-atomaire S-expressie in list-notatie is bijvoorbeeld (A B C); de listelementen worden van elkaar gescheiden door spaties. Om de relatie tussen de twee notaties te kunnen demonstreren moeten we het speciale atomaire symbool NIL invoeren, dat bij de list overeenkomt met de lege list. De list (A B C) komt dan overeen met (A.(B.(C.NIL))) en de list (A B (C D)) met (A.B.((C.D.NIL)).NIL))).

9.2.2 Operaties

We zullen de operaties hier eerst in een soort pseudo-LISP geven om dan daarna de eigenlijke LISP-notatie te bekijken.

LISP kent vijf basisfuncties op S-expressies. Deze functies zijn CAR, CDR, CONS, ATOM en EQ.

- CAR

Laat a een niet-atomaire S-expressie zijn (in punt-notatie) $a = (a_1.a_2)$, dan geldt:

$$CAR(a) = a_1$$

Voor de list-notatie geldt dat voor $l = (l_1 l_2 \ldots l_n)$

$$CAR(l) = l_1$$

- CDR

Als a een niet-atomaire S-expressie is (in punt-notatie) $a = (a_1 . a_2)$, dan geldt:

$$CDR(a) = a_2$$

Als $l = (l_1 l_2 ... l_n)$ een S-expressie in list-notatie is, dan geldt

$$CDR(l) = (l_2 ... l_n)$$

- CONS

Uit de twee S-expressies a_1 en a_1 wordt een nieuwe S-expressie gevormd:

$$CONS(a_1, a_2) = (a_1 . a_2)$$

Voor lists geldt dat als a een S-expressie is en l een list $l = (l_1 l_2 ... l_n)$, dat

$$CONS(a, l) = (a\ l_1 l_2 ... l_n)$$

OPMERKING

Er geldt:

$$CONS(CAR(a), CDR(a)) = a$$

$$CAR(CONS(a_1, a_2)) = a_1$$

$$CDR(CONS(a_1, a_2)) = a_2$$

(einde opmerking)

- ATOM

Deze functie levert het atomaire symbool T (voor true) op als het argument een atomaire S-expressie is en een F (voor false) als dit niet zo is.

- EQ

Deze functie levert T op als de twee argumenten atomaire S-expressies zijn en gelijk zijn en F als de twee atomaire S-expressies niet gelijk zijn (de functie is alleen gedefinieerd voor atomaire S-expressies als argumenten).

Naast deze functies kent LISP een soort conditional statement die we kunnen schrijven als

$$(p_1 \rightarrow e_1; p_2 \rightarrow e_2; ...; p_n \rightarrow e_n)$$

Hierin zijn $p_1, p_2, ..., p_n$ S-expressies of functies die een waarde T of F opleveren. De betekenis van deze constructie is

```
if p1 then e1
       else if p2 then e2
                  else .....
                       .....
                       else if pn then en fi
                  .....
            fi    .....
fi
```

Als geen der p_i's de waarde T oplevert is de constructie ongedefinieerd.

Alle constructies worden in LISP genoteerd als een list. Voor de constructies hierboven worden deze lists:

```
(CAR a)
(CDR a)
(CONS a1 a2)
(ATOM a)
(EQ a1 a2)
(COND (p1 e1)
      (p2 e2)
      .......
      (pn en))
```

In LISP komen ook getalwaarden voor. Operaties hierop zijn PLUS, DIFFERENCE, TIMES, DIVIDE, SUB1 ((SUB1 a) heeft als resultaat a-1) en nog enkele andere. Ook deze operaties worden geschreven als lists. Dus a+b*c wordt geschreven als (PLUS a (TIMES b c)).

Ook de logische operatoren AND, OR en NOT komen voor in LISP, evenals de relatie-operatoren LESSP (<), GREATERP (>), ZEROP (=0) en MINUSP (<0).

9.2.3 Programma's

Een LISP-programma is een functie die voor bepaalde argumenten geëvalueerd wordt. Het resultaat kan een list zijn.

In de wiskunde kunnen we een functie definiëren door bijvoorbeeld f(x,y) = x-y en de waarde van deze functie berekenen door aan te geven wat de waarden van de argumenten zijn: f(3,2). De functie heeft een naam gekregen, f, om er aan te kunnen refereren, zoals in f(3,2). Functies kunnen in de wiskunde ook gedefinieerd worden zonder een naam, door gebruik te maken van de zogenaamde lambda-notatie: λ((x,y) x-y). Toepassing van deze functie op de argumenten 3 en 2 wordt dan genoteerd als λ((x,y)x-y)(3,2). In LISP is deze notatie overgenomen en de functie ziet er dan uit als

```
(LAMBDA(X Y) (DIFFERENCE X Y))(3 2)
```

Algemeen:

(LAMBDA(formele$_1$...formele$_n$) functie-definitie)
(actuele$_1$...actuele$_n$)

Het kan echter zijn dat de functie-definitie recursief moet zijn, dus dat in deze definitie aan de functie zelf gerefereerd moet kunnen worden. Dit kan door de operator LABEL te gebruiken die twee argumenten heeft: de naam van de functie en zijn definitie. Stel dat we een functie willen maken die het eerste atoom uit een list moet opleveren. (Het eerste element van een list behoeft niet atomair te zijn.) In een informele notatie zou deze functie kunnen luiden: FIRST(X) = (ATOM(X)→X;
T→FIRST(CAR(X)))
Dit wordt nu:

```
(LABEL FIRST (LAMBDA(X)
         (COND((ATOM X) X)
           ((QUOTE T) FIRST(CAR X))))))
```

In deze functie geeft QUOTE aan dat het hier om de waarde T gaat en dat T niet optreedt als een soort variabele.

Nu de functie gedefinieerd is kunnen we deze functie laten uitvoeren (berekenen) door het LISP-systeem met bijvoorbeeld als argument de list ((A B) C). Om dat te bereiken wordt de functie en het argument aan de LISP-interpretator aangeboden door middel van

```
(EVALQUOTE(LABEL FIRST(LAMBDA...)))(((A B)C))
```

Een programma zelf heeft dus de vorm van een lambda-expressie (voorafgegaan door 'LABEL functienaam' als dat nodig is). De EVAL zorgt voor de evaluatie van de LAMBDA-expressie en de QUOTE zorgt voor het toepassen van QUOTE (zie hierboven) waar dat nodig is; vanaf hier zullen we QUOTE weglaten.

Wat we nu bekeken hebben wordt wel puur LISP genoemd. Dit is dus een subset van LISP die bestaat uit:
- de basisfuncties CAR, CDR, CONS, EQ en ATOM
- de besturingsstructuren COND, recursie en het samenstellen van functies
- lists
- LABEL en LAMBDA voor het vastleggen van de functies.

LISP kent echter meer constructiemogelijkheden, die vooral gericht zijn op het gemakkelijker hanteerbaar maken van de taal. We zullen enkele van deze constructies de revue laten passeren. De functie FIRST zoals we die hierboven gedefinieerd hebben kan maar één keer gebruikt worden. Willen we in een programma een functie meerdere keren gebruiken dan zou de functie een naam moeten krij-

gen om er aan te kunnen refereren. Dit kan niet met 'LABEL functie-naam' omdat deze constructie alleen bedoeld is voor lokale referentie in de functie-definitie. Voor globale referentie (die ook lokaal gebruikt kan worden) wordt de DEFINE-constructie gebruikt. Stel dat we als functie de faculteit (n!) willen definiëren. Dit kan met:

```
DEFINE((FACTORIAL (LAMBDA(N)
                  (COND((ZEROP N) 1)
                       (T (TIMES N FACTORIAL (SUB1 N)))))))
```

De definitie van een lambda-functie bestaat uit een samenstelling van functies. Deze definitie kan erg ingewikkeld worden door de recursie die hierin zal voorkomen. Als deel van zo'n definitie kan echter een PROG-constructie voorkomen, die de mogelijkheid biedt om constructies te realiseren die lijken op constructies in talen als Pascal en FORTRAN. De PROG-constructie heeft als vorm:

(PROG <lijst van lokale variabelen> <lijst van statements>)

De statements zijn S-expressies. De lokale variabelen kunnen waarden krijgen door een soort assignment (waarover we tot nu toe niet beschikten!):

(SET (QUOTE X) 4.2)

Hierdoor krijgt X de waarde 4.2. De QUOTE wordt weer gebruikt om aan te geven dat het om X zelf gaat. Als afkorting van het bovenstaande kan ook gebruik worden gemaakt van (SETQ X 4.2).

De statements in de lijst van statements zijn S-expressies, die sequentieel worden uitgevoerd. Een van de vormen is de 'assignment' die zojuist is geïntroduceerd. Statements kunnen een label krijgen waaraan gerefereerd kan worden in een sprongopdracht 'GO label'.

Met deze faciliteit kunnen iteraties gerealiseerd worden die in puur LISP niet mogelijk zijn. Laten we als voorbeeld nog eens de definitie van de faculteit nemen. Met de PROG-constructie kan een iteratieve versie als volgt gerealiseerd worden (waarbij de PROG dynamisch wordt afgebroken door RETURN):

```
DEFINE((
  (FACTORIAL (LAMBDA (N)
    (PROG (K)
      (SETQ K 1)
      LOOP (COND ((ZEROP N) (RETURN K)))
           (SETQ K (TIMES N K))
           (SETQ N (SUB1 N))
           (GO LOOP))))))
```

Het is mogelijk functies te laten optreden als argumenten van andere functies, waarbij het resultaat weer een functie is die dus toegepast kan worden op argumenten. Een voorbeeld hiervan is de functie MAPLIST in LISP die de vorm

MAPLIST (X FN)

heeft, waarbij X een list is en FN een functie is met één argument. Het resultaat van MAPLIST is een list die wordt gevonden door de functie FN toe te passen op achtereenvolgende delen van X.

MAPLIST kan gedefinieerd worden als:

```
DEFINE(((MAPLIST (LAMBDA (X FN)
                 (COND((NULL X) NIL)
                      (T (CONS (FN X)
                               (MAPLIST (CDR X) FN))))))))
```

Hierin is NULL een boolean functie die de waarde T oplevert als het argument NIL is en F als dit niet het geval is. Stel nu dat we de functie SQUARECAR hebben die als argument een list heeft en als resultaat een kwadraat van de CAR van die list. Dan levert

```
(LAMBDA (J) (MAPLIST J (FUNCTION SQUARECAR)))((1 2 3 4 5))
```

als resultaat op (1 4 9 16 25) (FUNCTION is een zelfde soort constructie voor functies als QUOTE voor variabelen).

9.3 SNOBOL

SNOBOL is ontwikkeld op het Bell Telephone Laboratorium in de Verenigde Staten in de jaren 1960-1969. De taal was oorspronkelijk bedoeld voor tekst-(string-)manipulatie maar is later uitgebreid met zodanige constructies dat de taal ook als algemene programmeertaal gebruikt kan worden (SNOBOL4). We zullen hier echter vooral de speciale faciliteiten voor tekstmanipulatie bekijken.

SNOBOL wijkt op een aantal punten af van programmeertalen als COBOL en FORTRAN:
- Allereerst natuurlijk de nadruk op het string-datatype.
- Daarnaast de mogelijkheid om nieuwe datatypen te introduceren.
- Tenslotte de mogelijkheid om, net als in LISP, programma's te laten uitvoeren die als data zijn opgebouwd of ingelezen.

9.3.1 Datatypen

Het voornaamste datatype is de string van karakters. Zo'n string kan op elke plaats in het programma ook optreden als de naam van een variabele en er kan dus een waarde aan worden toegekend. Variabelen worden dan ook niet gedeclareerd.

SNOBOL kent arrays van vaste lengte maar het type van de elementen kan variëren gedurende het programma. Array's worden geintroduceerd door bijvoorbeeld X = ARRAY(10). Zo'n declaratie kan

opgevat worden als de creatie van een pointer onder de naam X. Individuele elementen worden geselecteerd door bijvoorbeeld X<5>. Omdat het om een pointer gaat zullen na de assignment Y = X zowel X als Y verwijzen naar het array. Deze verwijzingen kunnen teniet worden gedaan door X en Y andere waarden te geven. Array's kunnen van willekeurige dimensie zijn.

Naast array's beschikt SNOBOL over tabellen (tables). Hierbij zijn de indices niet gehele getallen zoals bij array's, maar strings. De statement A = TABLE(30, 20) creëert in eerste instantie een tabel van dertig plaatsen. Een opdracht als A<'TYD'> = 25 zoekt in de tabel naar een index TYD en als deze nog niet is toegewezen, wordt er een nieuwe relatie (TYD, 25) vastgelegd. Als dit niet kan binnen de eerste derig gecreëerde plaatsen, wordt een volgend blok van twintig plaatsen gecreëerd.

Naast het stringtype kent SNOBOL nog het patroontype, dat van dezelfde vorm is als het stringtype. Het wordt als een apart type gezien omdat patroonherkenning en patroonsubstitutie in strings in SNOBOL zo'n voorname plaats innemen. Een patroon is de definitie van een klasse van strings met een gemeenschappelijke eigenschap. Het patroon legt die eigenschap vast. Een patroon kan op een aantal manieren opgebouwd worden uit patroonelementen. We komen hierop terug in 9.3.2.

SNOBOL kent de mogelijkheid om nieuwe datatypen te introduceren. Zo'n type heeft veel weg van een record. De definitie van een nieuw type is van de vorm

```
DATA('<name> (<item-list>)')
```

Bijvoorbeeld:

```
DATA('COMPLEX(RE, IM)')
```

en

```
DATA('PERSON(NAME, AGE, ADDRESS)')
```

Het refereren aan de componenten van de variabele X van het type COMPLEX gebeurt door RE(X) en IM(X).

9.3.2 Statements en operaties

De statements van SNOBOL kunnen grofweg in twee klassen ingedeeld worden: de assignment statement en de patroonherkenning- en patroonsubstitutie-statement. De assignment statement is van de vorm

```
<variable> = <expression>
```

Voor we ingaan op de tweede klasse statements bekijken we eerst de operaties op strings en patronen.

Twee strings kunnen geconcateneerd worden; de operator hiervoor is de betekenisvolle spatie: de blank. Zo levert

'PIET' 'PAALTJES'

de string 'PIETPAALTJES' op. Naast de concatenatie kent SNOBOL een groot aantal standaardfuncties voor strings, bijvoorbeeld:
- SIZE(<string>)
  Deze levert als resultaat de lengte van het argument op (SIZE('DRIE') = 4).
- DUPL(<string>, <integer>)
  Het resultaat van deze functie is de string die ontstaat door een aantal (aangegeven door het tweede argument) concatenaties van het eerste argument (DUPL('HA', 2) = 'HAHA').
- REPLACE(<string1>, <string2>, <string3>)
  In string1 wordt ieder karakter dat voorkomt in string2 vervangen door het overeenkomende karakter in string3 (REPLACE('HERHAAL', 'AH', 'EV') = 'VERVEEL').
- DIFFER(<string1>, <string2>)
  Een boolean functie die true is (en als nevenresultaat de lege string '' oplevert) als string1 en string2 niet gelijk zijn.
- IDENT(<string1>, <string2>)
  Deze functie is de 'inverse' van DIFFER.

De patroonherkenning-statement heeft de vorm

<string> <patroon>

waarin <string> een expressie is (bijvoorbeeld een variabele) die een string oplevert waarvan nagegaan wordt of <patroon> daarin als substring voorkomt.

De (patroonherkenning- en) patroonsubstitutie-statement heeft de vorm

<string> <patroon> = <substitutie-string>

De <substitutie-string> wordt in de <string> opgenomen op de plaats waar in deze string het patroon is gevonden.

Beide statements leveren als een soort neveneffect een boolean waarde op die gebruikt kan worden om in het programma een impliciete sprongopdracht te realiseren. We komen op dit punt later terug. Naast het zojuist genoemde neveneffect kunnen de statements nog als neveneffect hebben dat aan een variabele een stringwaarde wordt toegekend.

De patroonherkenning kan in twee 'toestanden' plaatsvinden: 'vast' of 'variabel'. Deze toestand wordt bepaald door de waarde van een systeemvariabele, die in het programma van waarde kan worden veranderd. We komen hier later op terug.

De herkenning in string S van patroon P kan beschreven worden in termen van de volgende pointers:

- LEFT
  Deze pointer wijst naar het karakter in S dat optreedt als het meest linkse karakter van de substring van S die onderzocht wordt.
- CURSOR
  Deze pointer wijst naar het meest rechtse karakter bij de patroonherkenning.

De patroonherkenning in S vindt plaats van links naar rechts. LEFT en CURSOR wijzen in het begin naar het meest linkse karakter van S. CURSOR 'loopt' naar rechts in de string (als de vorige karakters herkend zijn). Als er een herkenning plaatsvindt, wijzen LEFT en CURSOR naar het linker respectievelijk rechter karakter van de substring die herkend is. Als geen herkenning plaatsvindt zijn er twee voortzettingen mogelijk. De voortzetting hangt af van de toestand waarin de patroonherkenning plaatsvindt. In de vaste toestand blijft LEFT wijzen naar het meest linkse karakter van de string en als er geen herkenning plaatsvindt is het (neven-)effect dat de waarde false wordt opgeleverd. In de variabele toestand wordt het herkenningsproces herhaald met LEFT wijzend naar het volgende karakter van string S. De totale herkenning eindigt als het patroon wordt gevonden of als LEFT naar alle karakters van de string S heeft gewezen.

Het eenvoudigste patroon is een string. De statement

```
S 'JA'
```

is een patroonherkenning, waarbij 'JA' wordt gezocht in de string S. LEFT en CURSOR wijzen naar het eerste karakter van S. Is dit karakter een J dan wordt de CURSOR een karakter opgeschoven. Als de 'vaste toestand' heerst moet, wil herkenning plaatsvinden, S beginnen met JA.

We zullen nu andere mogelijkheden voor patronen bekijken. Een patroon kan uit elementen zijn opgebouwd. Bij de herkenning van een patroon speelt dan ook een derde pointer een rol: de NEEDLE. Deze pointer wijst naar het patroon-element dat nu in de patroonherkenning gebruikt wordt.

Als X en Y patronen zijn, dan is X Y het patroon dat ontstaat door X en Y te concateneren. Zo ook is X|Y een patroon; X|Y wordt herkend in string S als X of Y herkend wordt in S (hierbij speelt dus NEEDLE een rol). In de rij statements:

```
PAT = 'OU'|'AU'
S = 'PAUL'
S PAT = 'EE'
```

is de laatste statement een patroon-substitutie. Het resultaat van deze statements is dat S de waarde 'PEEL' heeft.

Voor het creëren van patronen staan ook een aantal functies ter

beschikking:

- ANY(<string>)
  Deze functie definieert een verzameling van patronen, waarbij ieder patroon overeenkomt met een van de karakters uit de argumentstring. ANY('ABC') heeft tot gevolg dat de CURSOR één plaats opschuift als er een A, een B of een C wordt herkend.
- NOTANY(<string>)
  Nu wordt de CURSOR een plaats opgeschoven als er een karakter wordt aangewezen dat niet voorkomt in de argumentstring.
- SPAN(<string>)
  Als deze functie wordt gebruikt wordt de CURSOR opgeschoven tot het eerste karakter dat niet voorkomt in de argumentstring.
- BREAK(<string>)
  Zoals NOTANY de tegenhanger is van ANY, zo is BREAK de tegenhanger van SPAN. De CURSOR wordt opgeschoven tot het eerste karakter dat voorkomt in de argumentstring.
- REM
  De CURSOR wordt opgeschoven tot het eind van de string.
- LEN(<integer>)
  Dit patroon komt overeen met iedere (sub)string van k karakters, als k het argument is van de functie. (De CURSOR wordt zonder meer over k plaatsen verschoven.)
- TAB(<integer>)
  De CURSOR wordt op het karakter geplaatst waarvan het argument de plaats aangeeft. Bijvoorbeeld: P TAB(10) 'PIET'
  Het patroon is een concatenatie van TAB(10) en 'PIET' en het patroon wordt herkend als direct na het tiende karakter de substring 'PIET' volgt.
- RTAB(<integer>)
  Deze functie is analoog aan TAB, alleen wordt nu de k-de plaats van rechts genomen als k het argument is.
- POS(<integer>)
  POS(k) is een patroon dat herkend wordt als de CURSOR op het k-de karakter gepositioneerd is.
- RPOS(<integer>)
  Hetzelfde resultaat als POS, alleen wordt nu gekeken naar de k-de plaats van rechts.

Bij patroonherkenning kan aan een variabele een waarde worden gegeven; deze waarde is de substring die overeenkomt met het herkende element uit het patroon.

VOORBEELD

```
P = ('A'|'I'|'O').X
S = 'PIET'
S P = 'E'
```

Deze statements hebben tot gevolg dat S de waarde 'PEET' krijgt en X de waarde 'I'.
(einde voorbeeld)

VOORBEELD

```
S SPAN(' ') BREAK(' ').X SPAN(' ')
```

Het patroon bestaat uit een aantal spaties, dan een aantal karakters ongelijk aan een spatie (noem deze substring T) en dan weer een aantal spaties. Als dit patroon herkend wordt krijgt de variabele X de waarde T.
(einde voorbeeld)

Deze vorm van assignment noemt men in SNOBOL de conditional value assignment.

Als een patroon 'JAN' gezocht wordt, kan via de bovenstaande assignment vastgelegd worden of dit patroon herkend is. Als het patroon niet herkend wordt, kan het toch nuttig zijn te weten hoe ver de herkenning wel geslaagd is. Dit kan via de immediate value assignment. We gaan er hier niet verder op in.

In de stukjes programma die we tot nu toe gezien hebben, worden de statements na elkaar uitgevoerd. Statements kunnen echter van een label voorzien worden en deże labels kunnen gebruikt worden om de volgorde van uitvoering te veranderen. Een onvoorwaardelijke sprongopdracht wordt bereikt via

```
<statement> : (<label>)
```

waarbij na uitvoering van deze statement de verwerking wordt voortgezet bij de statement met het aangegeven label.

Iedere statement in SNOBOL kan al dan niet succesvol uitgevoerd worden; het succesvol zijn of niet is een neveneffect, maar zeker bij patroonherkenning is duidelijk wat er bedoeld wordt. Het succesvol zijn of niet kan gebruikt worden door hierop te reageren via een sprongopdracht.

VOORBEELD

```
<statement> : S(<label 1>) F(<label 2>)
```

Als de statement succesvol uitgevoerd wordt, wordt er gesprongen naar label 1, anders naar label 2. In

```
<statement> : F(<label>)
```

wordt de volgende statement uitgevoerd als de uitvoering van de statement succesvol verloopt, anders wordt de verwerking voortgezet met de statement die wordt voorafgegaan door het genoemde label.
(einde voorbeeld)

VOORBEELD

```
LOOP ST ANY ('0123456789') = : S(LOOP)
```

Uitvoering hiervan heeft tot gevolg dat alle cijfers uit de string ST worden verwijderd.

9.3.3 Nog enkele taalelementen

Indirecte referentie

Een stringwaarde kan optreden als de naam van een variabele, waaraan dan weer een waarde kan worden toegekend. De overgang van waarde naar naam vindt plaats met behulp van de operator $:

```
P = 'WAARDE'
$P = 'MOOI'
```

Deze twee statements hebben tot gevolg dat de variabele, die ontstaat uit de waarde van P door toepassing van de operator $, de waarde 'MOOI' krijgt. De twee statements zijn equivalent aan

```
WAARDE = 'MOOI'
```

Subprogramma's

Subprogramma's in SNOBOL hebben de vorm van functies, die aangeroepen kunnen worden in expressies. Voor deze aanroep moet de functie gedefinieerd zijn. Deze definitie vormt geen syntactische eenheid en het is de verantwoordelijkheid van de programmeur om ervoor te zorgen dat bij uitvoering van een programma de statements van het hoofdprogramma en die van de functie gescheiden blijven. De functie wordt eigenlijk alleen door zijn eerste regel gedefinieerd:

```
DEFINE ('<functienaam> (<formele parameters>)
                  <lokale variabelen>', '<label>')
```

De label is de label van de eerste statement van de body. Dit label mag ontbreken; is dit het geval dan wordt de functienaam als zodanig genomen. Voorbeelden:

```
DEFINE('FUNCTIE(X, Y) A, B', 'BEGIN')
DEFINE('REV(S) T, U')
```

Zoals gezegd hoeft alleen deze regel vooraf te gaan aan de aanroep. Zo zou een programma met functie kunnen luiden:

```
    DEFINE('REV(S) T, U')
    A = INPUT                               : F(END)
    OUTPUT = 'INPUT IS: 'A
    OUTPUT = 'INPUT RESERVED:' REV(A): (END)
REV S LEN(1)                                : F(FRETURN)
    T = S
L   T LEN(1).U =                            : F(RETURN)
    REV = U REV                             : (L)
END
```

Het hoofdprogramma (met functieregel) bestaat uit de regels 1 tot en met 4 en regel 9. De body van de functie staat op de regels 5 tot en met 8. Als er geen invoer is eindigt het programma direct, anders

wordt in de vierde regel de functie geactiveerd.

Uit dit voorbeeld blijkt dat de functie verlaten kan worden via FRETURN (er wordt geen waarde opgeleverd) of via RETURN. De lokale variabelen en de functienaam (die ook als lokale variabele optreedt) worden geïnitialiseerd op de lege string.

De functie in het voorbeeld keert de volgorde van de karakters in een string om, mits het argument niet de lege string is.

Parameters in functies zijn value-parameters. Via indirecte referentie kan er echter voor gezorgd worden dat parameters zich gedragen als reference-parameters. Als P een parameter is die bij aanroep overeenkomt met de waarde 'Q', zal bij gebruik van $P in de functie gewerkt worden met de variabele Q.

9.4 SIMULA

De programmeertaal SIMULA (SIMulation LAnguage) is in 1965 ontwikkeld als een taal voor het programmeren en beschrijven van 'discrete event systems', zoals netwerken, communicatiesystemen, produktiesystemen, administratieve systemen, reserveringssystemen, enzovoorts; de taal wordt veel gebruikt voor grote simulatieprogramma's.

SIMULA is een uitbreiding van ALGOL 60; de belangrijkste uitbreidingen zijn het '*class*' en het '*coroutine*' concept, mogelijkheden voor tekst-manipulatie en invoer/uitvoer-faciliteiten. In 1970 werd de uiteindelijke versie (ook wel SIMULA 67 genoemd) gedefinieerd; hierbij kwamen enkele kleine wijzigingen ten opzichte van ALGOL 60 naar voren. We beschrijven hier enkele karakteristieken van SIMULA (67).

Bij het werken met problemen die een grote hoeveelheid details (gegevens, eigenschappen) bevatten is decompositie een belangrijk hulpmiddel. Door een groot probleem in stukken te splitsen wordt het overzichtelijk en hanteerbaar. In ALGOL 60 is de blokstructuur hiervoor geschikt (procedures). In SIMULA worden hier classes en coroutines (quasi-parallelle programma's) aan toegevoegd. We zullen deze twee concepten hier bespreken.

9.4.1 Classes

Een belangrijk concept in SIMULA is het 'object'; alle objecten kunnen door middel van een naam geaccesseerd worden. Een object is een exemplaar van een speciaal soort blok: een class. Van een gedeclareerde class mogen willekeurig veel exemplaren (elk met een eigen naam) bestaan; dat heten dan objecten van dezelfde class. Een

object kan een bepaalde reeks acties uitvoeren door middel van statements in het blok. Tevens heeft een object eigenschappen (zogenaamde attributen), die gerepresenteerd worden door lokale variabelen en parameters. De statements in het blok voeren operaties uit op die attributen. De class declaratie geeft aan, voor elk later te genereren object van die class, welke attributen relevant zijn en hoe het gedrag van zo'n object is; de class declaratie levert dus een patroon waar alle te genereren objecten van die class aan voldoen.

De meest eenvoudige vorm van een class declaratie is:

```
class <class identifier>;
begin <declaration of attributes>;
      <statements>
end
```

Bij de <declaration of attributes> worden lokale variabelen gedeclareerd; dat zijn variabelen van een standaardtype, arrays, procedures of objecten van een reeds gedefinieerde class.

VOORBEELD

Bij de simulatie van het scheepvaartverkeer in een haven spelen boten een belangrijke rol. Voor zo'n boot zijn fysieke zaken, zoals tonnage, lengte en lading belangrijk, maar ook het gedrag in de haven. Eenvoudig voorgesteld kan de definitie van de class luiden:

```
class boot;
begin integer tonnage, lengte, lading;
      'haven binnen varen';
      'vracht uitladen';
      'nieuwe vracht laden';
      'haven uit varen';
end
```

(einde voorbeeld)

Om in een SIMULA-programma een object van een gedeclareerde class te genereren wordt gebruikt het symbool <u>new</u> (de zogenaamde <object generator>).

Zo'n object wordt accesseerbaar en benoembaar door een toekenning aan een 'reference' variabele die gedeclareerd wordt door:

```
ref (<class identifier>) <list of identifiers>
```

De toekenning aan een reference variabele geschiedt met behulp van het symbool :-. Het genereren van een object en het toekennen aan een reference variabele is dus van de vorm:

```
<identifier> :- new<class identifier>
```

VOORBEELD
Na de declaratie van twee reference variabelen worden daar twee objecten van de class 'boot' aan toegekend:

```
ref(boot) Maria Jacoba, Chita;
Maria Jacoba :- new boot;
Chita :- new boot;
```

(einde voorbeeld)

In SIMULA hebben alle gedeclareerde variabelen een default initiële waarde. Boolean variabelen hebben default de waarde false, aritmetische variabelen de waarde nul, en reference variabelen de waarde none.

Als een object gegenereerd wordt, worden direct de statements van het blok uitgevoerd. Na afloop daarvan heet het object 'terminated' maar het bestaat nog wel. Na deze 'creatie' van het object wordt vervolgd met de volgende statement na de <object generation>.

Na toekenning van een object aan de reference variabele kan aan de attributen van het object gerefereerd worden door middel van de 'dot notatie': als A de naam van een attribuut is en X de naam van (de reference variabele voor) een object, dan is dat attribuut accesseerbaar met X.A. Met de 'dot notatie' kan aan attributen een waarde worden toegekend, of de waarde worden geïnspecteerd (dit kan natuurlijk niet indien de waarde van de reference variabele none is).

VOORBEELD

```
Maria Jacoba.tonnage := 256;
if Chita.lengte > 53 then .....;
totaal := Maria Jacoba.lading + Chita.lading;
```

(einde voorbeeld)

Bij een class declaratie kunnen ook formele parameters gespecificeerd worden. De parameteroverdracht is (anders dan in ALGOL 60) als volgt:
- standaardtypen (real, integer, character, boolean): 'by value';
- reference parameters, als zodanig gespecificeerd: 'by reference';
- arrays van een standaardtype: default 'by reference', indien in de value list opgenomen 'by value';

(N.B.: voor procedure-parameters geldt hetzelfde, en tevens:
- parameters van het type procedure worden 'by reference' overgedragen;
- elk van de parameters kan 'by name' overgedragen worden door ze expliciet in de name list op te nemen.).

De parameteroverdracht vindt plaats op het moment dat het object gegenereerd wordt.

VOORBEELD
Declaratie van een class voertuig met twee formele parameters en twee lokale variabelen; elk te genereren object zal dus vier attributen hebben.

```
class voertuig (bouwjaar, gewicht);
integer bouwjaar, gewicht;
begin boolean geregistreerd, stuk;
      .....
end;
```

Na declaratie van twee reference variabelen:

```
ref (voertuig) R14, Accord;
```

worden daar twee te genereren objecten aan toegekend:

```
R14 :- new voertuig (1980, 860);
Accord :- new voertuig (1979, 959);
```

Na de creatie van deze objecten kan aan de attributen gerefereerd worden, bijvoorbeeld:

```
if Accord.bouwjaar < 1981 then .....;
R14.geregistreerd := true;
Accord.stuk := false;
```

(einde voorbeeld)

Vaak zullen objecten van een verschillende class een aantal attributen gemeenschappelijk hebben, naast een aantal onderscheidende attributen. In SIMULA bestaat de mogelijkheid een class te declareren, en daarbinnen weer subclasses die de attributen van de omvattende class dan overnemen. Welke de omvattende class van een te declareren class is, wordt (eventueel) aangegeven met de zogenaamde 'prefix': de naam van die omvattende class. Aldus kan een hiërarchie van classes ontstaan, bijvoorbeeld:

```
class A   .....;
A class B .....;
B class C .....;
C class D .....;
```

Het totaal aantal attributen van een object uit een class is dus het aantal lokale variabelen en parameters in de class declaratie, plus alle lokale variabelen en parameters van (eventuele) omvattende classes. Uitvoering van de statements van het blok (bij creatie van een object) wordt voorafgegaan door uitvoering van de statements van het blok van de omvattende class, enzovoorts.

VOORBEELD
Subclasses van de class voertuig kunnen als volgt worden gedeclareerd:

```
voertuig class vrachtauto (laadvermogen);
real laadvermogen; {in tonnen}
begin boolean tientonner;
      real lading;
      ... if laadvermogen ≧ 10 then tientonner := true; ...
end;
voertuig class personenauto (maxsnelheid, aantzitplaatsen);
real maxsnelheid; integer aantzitplaatsen;
begin ...
end;
voertuig class takelwagen;
begin ref (voertuig)pechvogel;
      procedure slepen (PV); ref (voertuig)PV;
      begin if PV.stuk then pechvogel :- PV else 'valsalarm'
      end
end
```

Na declaratie en object-generatie (waarbij de parameters van de omvattende class eerst gegeven worden, enzovoorts):

```
ref (vrachtauto)V;
ref (personenauto)P;
ref (takelwagen)T;
V :- new vrachtauto (1976, 4000, 28.3);
P :- new personenauto (1981, 970, 156.0, 5);
T :- new takelwagen (1976, 4050)
```

kunnen de attributen geaccesseerd worden:

```
V.geregistreerd := true;
V.lading := 16.0;
if V.tientonner then .....;
P.stuk := true;
T.slepen(P);
```

Het object pechvogels in de procedure 'slepen' van de class takelwagen is oorspronkelijk none. De takelwagen kan elk voertuig slepen: een vrachtauto, een personenauto of een takelwagen. Door aanroep van de procedure 'slepen' krijgt pechvogel een waarde (de actuele parameter P wordt 'by reference' overgedragen).
(einde voorbeeld)

Toekenning aan een reference variabele gebeurt met de new statement, maar ook de toekenning van de waarde none is toegestaan; een derde mogelijkheid is de toekenning van de waarde van een andere reference variabele, waarvan de waarde dan wel van de

juiste class of subclass moet zijn. Bijvoorbeeld:

```
ref(voertuig)C1; C2;
C1 :- none;
C2 :- new voertuig (1976, 1400);
C1 :- new personenauto (1974, 1100, 144.3, 4);
C2 :- C1;
```

Waarden van reference variabelen kunnen met elkaar vergeleken worden. Er zijn vier operatoren (X en Y zijn reference variabelen, N is de naam van een class):

X == Y	: true d.e.s.d. als X en Y naar hetzelfde object refereren, of beide de waarde none hebben;
X =/= Y	: not (X == Y);
X is A	: true d.e.s.d. als X refereert naar een object van class A, false anders;
X in A	: true als X refereert naar een object van class A of van een subclass van A, false anders.

9.4.2 Coroutines

Als strikt sequentiële programmeertaal heeft SIMULA toch mechanismen die ons in staat stellen parallelle processen te simuleren. Deze quasi-parallelle programma's (coroutines) verdelen de rekentijd van een sequentiële processor zo dat steeds slechts één component actief is terwijl toch, door wisseling van actieve component, alle componenten voortgang maken. Het creëren van zulke quasi-parallelle componenten gebeurt door het genereren van objecten. Indien een object gecreëerd wordt, wordt het blok van de desbetreffende class voor dat object uitgevoerd; daarna wordt vervolgd met de statement na die <object generation>.

Indien nu in zo'n blok statements voorkomen die bewerkstelligen dat met een andere component (bijvoorbeeld het genererende programma of een ander object) wordt vervolgd, ontstaat de mogelijkheid tot wisseling van actieve component. In SIMULA kennen we:
- detach: vervolg met de statement na de <object generation>;
- resume(x): vervolg met component x met de statement na de laatst uitgevoerde detach of resume statement.

Een programma dat eerst een aantal componenten creëert van de classes A, B, C, ..., en vervolgens met deze componenten op een quasi-parallelle manier gaat samenwerken, zal de volgende structuur hebben:
- Elk van de classes A, B, C zal de vorm hebben:

```
class A ...;
begin <declaraties>;
      <statements>;
       detach;
      <statements>; {hier kunnen ook detach en resume
                     statements in voorkomen}
end;
```

- Het programma zelf zal de vorm hebben:

```
begin class A ...;
      class B ...;
      class C ...;
      ...
      'creëer componenten';
      resume(component);
end;
```

VOORBEELD

Een programma dat de 'samenwerking' van N tandwielen simuleert, door elk van de tandwielen, in de juiste volgorde, steeds één positie te laten verspringen.

```
begin integer N;
      N := ...;
      begin class tandwiel (m); integer m;
            begin ref(tandwiel) volgende;
                  procedure verspring1positie; ...;
                  detach;
                  while true do
                     begin verspring1positie;
                           resume(volgende);
                     end
            end;
            ref(tandwiel) array t[1:N];
            integer i;
            for i := 1 step 1 until N do t[i] :- new tandwiel(i);
            for i := 1 step 1 until N-1 do
                     t[i].volgende :- t[i+1];
            t[N].volgende :- t[1];
            resume (t[1]);
      end;
end
```

(einde voorbeeld)

9.4.3 Enkele opmerkingen

Bij het gebruik van SIMULA zijn twee – reeds gedefinieerde – classes standaard beschikbaar. Ze zijn zonder declaratie te gebruiken, alleen moet de naam als <prefix> voor het programma aangegeven worden. De class SIMSET biedt mogelijkheden voor het werken met 'circular double linked lists'; in SIMULA worden ze 'sets' genoemd en een element kan in een set ten hoogste één keer voorkomen.

Operaties op sets zijn o.a.:
- het verwijderen van elementen;
- het toevoegen van elementen;
- de opvolger/voorganger in de lijst van een element opvragen;
- het aantal elementen opvragen;
- opvragen of de set leeg is, en
- het leeg maken van een set.

De class SIMULATION biedt mogelijkheden voor het werken met 'discrete event' simulatie. De class SIMSET is een <prefix> van SIMULATION zodat deze als vanzelf beschikbaar is. Bij simulatie zijn vaak een aantal processen belangrijk waarop wacht- en voortgangsoperaties worden uitgevoerd. Voorbeelden van dit soort operaties zijn:
- wacht een bepaalde tijd,
- het (tijdelijk) stilleggen van een proces, en
- het (opnieuw) activeren van een proces.

Tot slot geven we voor de volledigheid de verschillen tussen ALGOL 60 en SIMULA nog eens weer.
SIMULA bevat als uitbreidingen van ALGOL 60:
- het class, reference en object concept;
- parameteroverdracht 'by reference';
- het type karakter, tekstmanipulatie en invoer/uitvoer-faciliteit;
- coroutines.

De veranderingen ten opzichte van ALGOL 60:
- parameteroverdracht is niet standaard 'by name';
- alle variabelen worden 'default' geïnitialiseerd;
- geen own concept;
- geen type string, maar het 'text' concept.

9.5 APL

APL (A Programming Language) is oorspronkelijk door K. Iverson ontworpen als een notatie om algoritmen in uit te drukken. In 1966 is voor het eerst een wat gewijzigde versie onder de naam APL/360 geïmplementeerd; latere implementaties waren meestal op APL/360 gebaseerd. We richten ons hier op deze versie van APL. Twee eigenschappen onderscheiden APL van de meer conventionele programmeertalen:

1. Het is een interactieve taal, dat wil zeggen dat de programmeur via een terminal met het systeem (bijvoorbeeld APL/360) communiceert. De programmeur typt een opdracht en het systeem voert die meteen uit. Daarna kan de volgende opdracht ingetypt worden, die weer direct uitgevoerd wordt, etc.
2. De elementaire eenheid van data is het array. De operaties werken op arrays (van alle mogelijke dimensies) en leveren arrays als resultaat op. Dit is een groot verschil met andere talen.

De notatie bestaat uit een bijzonder uitgebreide set van machtige operatoren, en een regel van een paar karakters, ingetypt via een terminal, kan een grote hoeveelheid rekenwerk inhouden. Omdat de notatie echter zeer cryptisch is, zijn programma's in APL wel haast onleesbaar; de compactheid leidt tot een soort geheimschrift, dat teken voor teken ontcijferd moet worden.

De variabelen in APL zijn arrays van het type 'number' of 'character'; dat type is dynamisch veranderlijk, zodat een variabele bijvoorbeeld op het ene moment als een vector van 'numbers' gebruikt kan worden, en op het andere moment als een vijf-dimensionale matrix van 'characters'. Elke operator is een functie die een waarde aflevert. De operatoren kunnen samengesteld worden, zodat zeer machtige expressies ontstaan. Binnen een expressie worden de operaties van rechts naar links uitgevoerd; deze regel kan door middel van haakjes doorbroken worden.

Een APL-programma bestaat uit een hoofdprogramma en een aantal procedures of functies (subprogramma's); er is geen blokstructuur.

Procedures hebben 0, 1 of 2 parameters die by value overgedragen worden bij aanroep, en 0 of 1 resultaat-parameter. Aangezien deze parameters arrays zijn is dat geen beperking. Een procedure kan een aantal lokale variabelen declareren; verder zijn alle globale variabelen accesseerbaar. Variabelen hoeven in principe niet gedeclareerd te worden; alleen formele parameters en lokale variabelen wel, om de 'scope' vast te leggen. Een procedure kan aangeroepen worden in een procedure of direct door de naam (eventueel met parameters) in te typen via de terminal.

De enige controlestructuur is de goto statement, die een regel-

nummer (elke regel bevat één statement die van het systeem een nummer krijgt) of een label aanduidt.

De uitvoering van een APL-programma verloopt als volgt:

- de programmeur creëert een aantal procedures;
- vervolgens wordt het programma regel voor regel ingetypt, waarbij elke regel direct uitgevoerd wordt.

9.5.1 Data

Variabelen en data zijn arrays van het type 'number' of 'character'. Om aan variabelen waarden toe te kennen (assignment : ←) wordt een vector van 'numbers' genoteerd als een reeks getallen gescheiden door spaties, en een vector van 'characters' als een karakter string. Arrays van hogere dimensie kunnen door operatoren uit vectoren gecreëerd worden. Via indices kunnen afzonderlijke delen of componenten van arrays geaccesseerd en veranderd worden. Bijvoorbeeld:

Als A een 3 * 4 matrix is dan:

- A[2; 3] accesseert de component in de tweede rij en de derde kolom;
- A[1;] accesseert de eerste rij;
- A[3; 1] ← 7 kent aan A[3; 1] de waarde 7 toe;
- A[1;] ← 7 kent aan elke component van de eerste rij de waarde 7 toe;
- A[1 2; 5 6] ← 7 kent aan alle componenten A[1; 5], A[1; 6], A[2; 5], A[2; 6] de waarde 7 toe.

VOORBEELD

```
      N ← 'APPELS'                  % assignment opdracht
      N[4]                          % vierde component
   E                                % resultaat
      N[1 3 5]                      % drie componenten
   APL                              % resultaat
```

9.5.2 Operaties

Elke operatie heeft arrays als operanden en levert een array als resultaat af. Indien beide arrays niet even groot zijn, wordt de kortste eenvoudigweg verlengd door herhaling. Een operatie met 1 operand wordt monadisch genoemd, een operatie met 2 operanden dyadisch; veel operatoren kunnen zowel monadisch als dyadisch gebruikt worden, soms met heel andere betekenissen. In het volgende worden een aantal van de operaties besproken.

a. Aritmetische operatoren

Optelling, aftrekking, vermenigvuldiging (×), deling, machtsverheffing (*) en ook absolute waarde, logaritme, worteltrekking, maximum, minimum, faculteit, sinus, cosinus en andere trigoniometrische functies kunnen niet alleen op getallen (of paren getallen), maar ook op arrays (of paren arrays van dezelfde grootte) worden toegepast. In dat laatste geval worden de operaties op elk element (of alle paren elementen) van de arrays uitgevoerd en het resultaat zal dan een array zijn van dezelfde grootte.

Voor de exacte notatie van de operatoren wordt verwezen naar de APL/360 reference manual.

VOORBEELD

```
      A ← 1 2 3 4                % assignment opdracht
      B ← 6 4 2 0                % assignment opdracht
      A + B                      % opdracht tot optelling
7 6 5 4                          % resultaat
      A ⌈B                       % opdracht tot maximum bepaling
6 4 3 4                          % resultaat: elke component is het
                                 % maximum van de twee corresponde-
                                 % rende componenten
```

b. Relationele en logische operaties

Logische waarden worden gerepresenteerd door 1 (true) en 0 (false), zodat de relationele en logische operaties te beschouwen zijn als speciale aritmetische operatoren; ze kunnen dan ook door elkaar gebruikt worden. De operaties gelijk, niet-gelijk, kleiner dan, groter dan, kleiner of gelijk, groter of gelijk, <u>not</u>, <u>and</u>, <u>or</u>, <u>nand</u> en <u>nor</u> kunnen ook weer op arrays toegepast worden.

VOORBEELD

```
      C ← A ≦ B                  % Elk paar componenten wordt
      C                          % met elkaar vergeleken
1 1 0 0                          % resultaat
      D ← 1 0 1 0                % assignment
      C ∧ D                      % opdracht tot conjunctie
1 0 0 0                          % resultaat
```

c. Array modificatie

Een aantal operatoren voeren operaties uit op arrays, zonder ze van waarde te veranderen.

De modanische operator 'ρ' retourneert een vector met als componenten het aantal waarden behorende bij iedere subscript. De dyadische operator 'ρ' maakt van een array als tweede operand een array van de vorm zoals aangegeven door de eerste operand.

VOORBEELD

Als A een 3 * 4 matrix is dan:

```
    ρA                          % aantal comp. per subscript
 3 4                            % resultaat
    ρρA                         % aantal dimensies
 2                              % resultaat
    B ← 1 2 3 4 5 6             % assignment opdracht
    2 3 ρ B                     % van vector een matrix maken
 1 2 3                          % resultaat
 4 5 6
```

(einde voorbeeld)

De monadische operator ',' maakt van een array van willekeurige dimensie een vector, waarbij de componenten rij voor rij achter elkaar geplaatst worden. De dyadische operator ',' concateneert twee vectoren tot één.

VOORBEELD

```
Als  A = 1 2 3
         4 5 6
dan: A ← , A                    % monadisch
     A
 1 2 3 4 5 6
     A, 7 8                     % dyadisch
 1 2 3 4 5 6 7 8
```

(einde voorbeeld)

Andere operaties zijn bijvoorbeeld: matrix transponeren, vector roteren, uitbreiding, inkorten. Ook is er een operator die voor een vector bepaalt: de rij indices van de plaatsen waar de maximale waarde staat; deze kan gebruikt worden bij het sorteren van arrays.

d. Generatoren

De monadische operator 'ι' geeft een vector van opeenvolgende natuurlijke getallen met als laatste de waarde van de operand (een skalar). Als dyadische operator geeft A ι B, met A en B vectoren, voor elke componentwaarde van B de kleinste index van A waar die waarde voorkomt. Als een componentwaarde niet voorkomt, wordt (de grootste index +1) gegeven.

VOORBEELD

```
    ι7                          % genereer 1 tot en met 7
 1 2 3 4 5 6 7                  % resultaat
    A ← 2 3 5 7                 % assignment opdracht
    A + ι4                      % expressie
 3 5 8 11                       % resultaat
    A ι 1 2 3                   % dyadisch
 5 1 2                          % resultaat
```

(einde voorbeeld)

Een zeer belangrijke operatie is reductie (/). Een van de dyadische aritmetische, relationele of logische operatoren wordt toegepast op alle componenten van een vector. De operator wordt toegepast alsof deze tussen alle componenten van de vector stond. Bij meerdimensionale arrays vindt de operatie plaats op de laatste subscript, tenzij de bedoelde subscript is aangegeven.

VOORBEELD

```
      A ← 1 2 3 4             % assignment opdracht
      +/A                     % alle componenten optellen
10                            % resultaat
      B ← 2 2 ρ A             % matrix maken
      ⌈/[1]B                  % maximum van kolommen bepalen
3 4                           % resultaat
      C ← 0 1 1 0             % assignment opdracht
      ∨/C                     % of reductie
1                             % resultaat (true)
      ∧/C                     % and reductie
0                             % resultaat (false)
      ∨/(1 2 3 4 = 3)         % is ten minste één van de
                              % componenten gelijk aan 3?
1                             % resultaat (true)
```

(einde voorbeeld)

Andere opdrachten zijn inprodukt en uitprodukt.

e. Invoer/uitvoer

Uitvoer geschiedt op twee manieren:

- Als een expressie door de programmeur ingetypt is, zal die na evaluatie vanzelf uitgevoerd worden.
- Als een assignment gebeurt aan het symbool '⎕' wordt de berekende waarde op de terminal afgedrukt.

Invoer maakt ook gebruik van het symbool '⎕'. Als dit in een expressie voorkomt, wordt bij evaluatie van de expressie gestopt, en gewacht op invoer van de terminal. Zodra die verschenen is wordt de evaluatie vervolgd met die betreffende waarde op de plaats van het symbool '⎕'.

f. Sprongopdrachten

Elke statement in een procedure-definitie krijgt een regelnummer, beginnend bij 1. De programmeur kan statements van een label voorzien. De goto statement (notatie →) kan de volgorde van executie wijzigen; er wordt vervolgd met de statement met het regelnummer (bijvoorbeeld in de vorm van een expressie) of met het label dat na deze pijl staat.

Conditionele sprongen zijn mogelijk. Als de expressie de waarde 0 oplevert, wordt de procedure verlaten.

9.5.3 Procedures

Procedure-definitie begint met het symbool '∇', eventueel gevolgd door de resultaat-parameter en het symbool '←', eventueel gevolgd door een tweede parameter, gevolgd door de procedure-naam, eventueel gevolgd door de eerste parameter. Daarna kan eventueel de declaratie van lokale variabelen volgen, door middel van een opsomming van namen, elk voorafgegaan door een puntkomma. De body wordt beschreven door een aantal statements, afgesloten door het symbool '∇'. Daarna kan de procedure aangeroepen worden.

VOORBEELD
Eerst wordt een procedure MOD zonder parameters gedefinieerd die N1 mod N2 berekent en die waarde aan N3 toekent. Vervolgens een procedure GGD met twee parameters, N1 en N2, en een resultaat-parameter K, die ggd(N1, N2) berekent; deze maakt gebruik van een lokale variabele N3.

```
      ∇MOD                     % procedure MOD
  [1] N3 ← N1                  % assignment opdracht
  [2] → (N3 < N2)/0            % als N3 < N2: sprong nr. regel 0
                               % (exit)
  [3] N3 ← N3 - N2             % assignment opdracht
  [4] → 2                      % sprong naar regel 2
  [5] ∇                        % einde definitie MOD
      ∇K ← N1 GGD N2; N3       % procedure GGD
  [1] MOD                      % aanroep MOD
  [2] → (N3 = 0)/E             % als N3 = 0: sprong nr. label E
  [3] N1 ← N2                  % assignment opdracht
  [4] N2 ← N3                  % assignment opdracht
  [5] → 1                      % sprong naar regel 1
  [6] E: K ← N2                % assignment opdracht
  [7] ∇                        % einde definitie GGD
      N1 ← 27                  % assignment opdracht
      N2 ← 5                   % assignment opdracht
      MOD                      % aanroep MOD
      N3                       % resultaat van MOD
  2                            % klopt
      12 GGD 21                % aanroep GGD
  3                            % ggd (12, 21) = 3
```

9.5.4 Voorbeelden

1. Een procedure om een vector A van getallen te sorteren in niet-dalende volgorde. Het resultaat wordt in de resultaat-parameter B afgeleverd.

```
    ∇B ← SORT A
[1] B ← 0ρ0                     % initialisatie van B: lengte 0
[2] → (0=ρA)/0                  % als lengte A = 0: exit
[3] MINS ← A = (⌊/A)            % ⌊/A bepaalt het minimum van A;
                                % die waarde wordt vergeleken met
                                % A en dit levert een logische
                                % vector, met enen op de plaats(en)
                                % van dat minimum
[4] B ← B, MINS/A               % MINS/A laat van A die componen-
                                % ten over waar de component van
                                % MINS 1 is. Deze worden met B
                                % geconcateneerd.
[5] A ← (~ MINS)/A              % ~ MINS staat voor not MINS;
                                % van A blijven dus die componen-
                                % ten over waar de component van
                                % MINS 0 is.
[6] → 2                         % sprong naar regel 2
[7] ∇                           % einde definitie SORT
    A ← 6 2 8 3 5               % assignment
    SORT A                      % aanroep van SORT
2 3 5 6 8                       % resultaat
```

2. Een procedure om, voor een gegeven waarde N, de eerste N priemgetallen te bepalen. Het resultaat wordt in de resultaat-parameter R afgeleverd.

```
    ∇R ← PRIEM N; T             % lokale variabele T
[1] R ← 2, T ← 3                % T krijgt de waarde 3
                                % R krijgt de waarde 2, geconcate-
                                % neerd met 3.
[2] → (N=ρR)/0                  % als de lengte van R=N: exit
[3] T → T + 2                   % T krijgt als waarde: het volgen-
                                % de oneven getal
[4] → (V/0=R/T)/3               % R/T levert een vector van resten
                                % van deling van T door alle com-
                                % ponenten R; alle componenten van
                                % deze vector worden met 0 verge-
                                % leken, dit levert een logische
                                % vector met enen op de plaats(en)
                                % waar 1 stond(en). De or reductie
                                % onderzoekt of er nullen waren;
                                % zo ja, sprong naar regel 3
[5] R ← R, T                    % de waarde van T (priem) wordt
                                % geconcateneerd met R
[6] → 2                         % sprong naar regel 2
[7] ∇                           % einde definitie van PRIEM
    PRIEM 4                     % aanroep van PRIEM
2 3 5 7                         % eerste vier priemgetallen
    X ← PRIEM 8                 % aanroep van PRIEM; resultaat
    X                           % wordt aan X toegekend
2 3 5 7 11 13 17 19             % resultaat
```

9.5.5 Slotopmerkingen

De syntactische structuur van APL is zeer eenvoudig en kan snel geleerd worden. De hoeveelheid symbolen echter, vele met een monadische en een dyadische functie, vergt wat meer moeite. De kracht van APL ligt toch in deze set van operatoren, die alle op arrays werken. Voor matrixwerk kan gebruik van APL zeker zijn nut hebben. In het algemeen geldt dat op allerlei mogelijke manieren met arrays gemanipuleerd kan worden. De operatoren kunnen samengesteld worden gebruikt, zodat de expressies bijzonder machtig kunnen zijn. Dat leidde tot het fenomeen 'one-liners', waarbij zeer complexe berekeningen als een programma van één regel geschreven worden. Programma's kunnen zó kort worden, dat het moeilijk is ze te ontcijferen.

Als interactieve taal bevat APL vele constructies voor 'samenwerking' met het APL-systeem (eigenlijk een verlengstuk van het operating system); bijvoorbeeld het creëren van een 'workspace' voor de gebruiker, die dan aldaar aanwezige programma's direct kan activeren. Deze faciliteiten maken een wezenlijk deel uit van de mogelijkheden die APL biedt, en dragen veel bij aan de populariteit die APL geniet onder de liefhebbers van interactief computergebruik.

10 ONTWIKKELINGEN

10.1 INLEIDING

In dit hoofdstuk komen enkele losse onderwerpen aan de orde die wellicht een plaats hadden kunnen krijgen in hoofdstuk 7. Dat zij hier opgenomen zijn heeft als voornaamste reden dat wij ons in hoofdstuk 7 hoofdzakelijk beperkt hebben tot constructies die in de hoofdstukken 8 en 9 aan de orde kwamen.

10.2 GUARDED COMMANDS, EN HET FORMEEL AFLEIDEN VAN PROGRAMMA'S

In 1975 publiceerde E.W. Dijkstra een eerste artikel over 'guarded commands'. Het is een notatie, die ons in staat stelt programma en correctheidsbewijs simultaan te ontwerpen. De semantiek van de – enkele – constructies is vastgelegd in predicaten-transformaties en daardoor is deze semantiek goed bruikbaar bij het afleiden van een programma bij een gegeven beginpredicaat en eindpredicaat (zie axiomatische semantiek, § 4.4).

Predicaten zijn uitspraken over de toestand. In het volgende staan P, Q en R voor predicaten die gedefinieerd zijn op de gehele toestandsruimte. Bij een predicaat hoort een verzameling toestanden waarvoor de uitspraak 'waar' is. Er zijn twee speciale predicaten: T (true) is waar voor alle toestanden, F (false) is voor geen enkele toestand waar.

Bij het ontwerp zijn we geïnteresseerd in een bepaalde eindsituatie, dat wil zeggen een toestand waar een bepaald predicaat 'waar' is. Het is de bedoeling een aantal toestandstransformaties te beschrijven zo dat, uitgaande van de toestand waarin het beginpredicaat geldt, door de successieve toestandstransformaties een toestand bereikt wordt waar het eindprodukt geldt. Die toestand, waar het eindpredicaat geldt, is dus het te bereiken doel, en hieruit

blijkt al dat programmeren doelgericht is. Daarom wordt voor de constructies met 'guarded commands' de semantiek beschreven in termen van *'weakest preconditions'*. Voor een constructie (een statement of reeks statements) S en een eindpredicaat R is de 'weakest precondition', wp(S, R), het zwakste predicaat voor de begintoestand zo dat uitvoering van S gegarandeerd binnen eindige tijd leidt tot een eindtoestand waar R geldt. Het woord 'zwakste' is essentieel: hoe zwakker een predicaat, des te groter het aantal toestanden waarin het waar is, en des te groter de kans dat de begintoestand hierbij behoort.

VOORBEELD
wp ("i := i+1", i ≦ 10) = (i ≦ 9), want als i ≦ 9 heeft uitvoering van "i := i+1" tot gevolg dat i ≦ 10 is, en als i > 9 is, zal achteraf zeker niet gelden i ≦ 10.
(einde voorbeeld)

Het ontwerpen van een programma voor beginpredicaat P en eindpredicaat Q komt nu neer op het ontwerp van een constructie S zo dat P => wp(S, Q), dat wil zeggen uitgaande van Q wordt een constructie S (eventueel samengesteld uit S_0; S_1; ...; S_n) gezocht zó dat de begintoestand, waarin P geldt, één van de toegestane toestanden is die bij uitvoering van S tot een eindtoestand leiden waarin Q geldt.

Weakest preconditions hebben de volgende eigenschappen:
1. wp(S, F) = F
2. Als P => Q dan: wp(S, P) => wp(S, Q)
3. (wp(S, P) <u>and</u> wp(S, Q)) = wp(S, P <u>and</u> Q)
4. (wp(S, P) <u>or</u> wp(S, Q)) => wp(S, P <u>or</u> Q)

De semantiek van een constructie S wordt nu beschreven in algemene 'weakest preconditions', zodat voor elk eindpredicaat R wp(S, R) af te leiden is.

1. De semantiek van de lege statement 'skip' is:
   wp('skip', R) = R
2. De semantiek van de assignment statement "x := E" is:
   wp("x := E", R) = R_E^x
   waarbij R_E^x staat voor R met daarin x vervangen door E.
   Voorbeeld
   wp("x := 5", x=5) = (5=5) = T
   wp("x := x↑2", x↑4 = 256) = ((x↑2)↑4 = 256) = (x↑8 = 256)
3. De semantiek van de concatenatie ';' is:
   wp("S_1; S_2", R) = wp("S_1", wp("S_2", R))

Voorbeeld
wp("x := 2; y := 2 * x", y = 4) =
wp("x := 2", wp("y := 2 * x", y = 4)) =
wp("x := 2", 2 * x = 4) =
2 * 2 = 4 = T

4. De alternatieve constructie wordt genoteerd als:
<u>if</u> $B_1 \rightarrow S_1 \square B_2 \rightarrow S_2 \square \ldots \square B_n \rightarrow S_n$ <u>fi</u>
De condities B_i heten guards. De constructie $B_1 \rightarrow S_1$, $B_2 \rightarrow S_2$, ... $B_n \rightarrow S_n$ vormen samen de zogenaamde 'guarded command set'. Een statement (eventueel reeks statements) S_i kan uitgevoerd worden als de bijbehorende guard true is. Meerdere guards kunnen true zijn; dan wordt een van de bijbehorende statements uitgevoerd. Als geen guard true is wordt dit als een fout beschouwd en wordt uitvoering van het programma gestopt.

Laat BB = B_1 <u>or</u> B_2 <u>or</u> ... <u>or</u> B_n, dan
wp("<u>if</u> $B_1 \rightarrow S_1 \square \ldots \square B_n \rightarrow S_n$ <u>fi</u>", R) =
BB <u>and</u> $(\forall_i : 1 \leq i \leq n: B_i \Rightarrow wp(S_i, R))$.
Hierbij betekent de eerste term dat ten minste één van de guards true moet zijn, en de tweede term dat voor elke guard die true is de bijbehorende statement leidt tot een toestand waar R geldt.

Voorbeeld
wp("<u>if</u> a ≦ b → m := b □ b ≦ a → m := a <u>fi</u>", m = max(a, b))=
(a ≦ b <u>or</u> b ≦ a) <u>and</u> (a ≦ b => wp("m := b", m = max(a, b))) <u>and</u>
(b ≦ a => wp("m := a", m = max(a, b))) =
true <u>and</u> (a ≦ b => b = max(a, b)) <u>and</u> (b ≦ a => a = max(a, b))=
true <u>and</u> true <u>and</u> true = T.

5. De repetitieve constructie wordt genoteerd als:
<u>do</u> $B_1 \rightarrow S_1 \square B_2 \rightarrow S_2 \square \ldots \square B_n \rightarrow S_n$ <u>od</u>
De repetitie eindigt zodra geen enkele guard true is. In elke slag wordt voor een van de guards die true zijn de bijbehorende statement uitgevoerd. De semantiek van deze repetitie is zeer gecompliceerd. Uitvoering van de repetitieve constructie is echter te beschouwen als een aantal malen een alternatieve constructie (met dezelfde 'guarded command set') uit te voeren tot ¬ BB geldt!

Daarom volgt hier de '*invariantie stelling* voor repetities' die de 'weakest precondition' voor de repetitie beschrijft door middel van een invariant P, een variante functie t, en de 'weakest precondition' voor de alternatieve constructie met dezelfde 'guarded command set'.

Als (P and BB) => wp("if $B_1 \rightarrow S_1$ □ ... □ $B_n \rightarrow S_n$ fi", P)
en (P and BB) => $t \geq 0$
en (P and BB and $t \leq to + 1$) =>
wp("if $B_1 \rightarrow S_1$ □ ... □ $B_n \rightarrow S_n$ fi", $t \leq to$)
dan: P => wp("do $B_1 \rightarrow S_1$ □ ... □ $B_n \rightarrow S_n$ od", P and ¬ BB).

In woorden: de guards zorgen ervoor dat een uitvoering van de alternatieve constructie de invariant P niet verstoort. Daarom zal na afloop van de repetitie P gelden en tevens ¬ BB. Verder geldt dat elke slag van de repetitie de waarde van t vermindert. Aangezien alleen de laatste slag van de repetitie $t < 0$ kan maken, zal de repetitie slechts een eindig aantal malen de waarde van t kunnen verminderen opdat $t \geq 0$ blijft. De repetitie is dus eindig.

Het is relevant hier op te merken dat een 'guarded command set' non-determinisme toestaat. Meerdere guards kunnen true zijn, en daaruit wordt er steeds één gekozen. De volgorde en selectie van statements hoeft dus niet bij voorbaat vast te liggen, hetgeen het afleiden van repetitie en alternatieve constructies eenvoudiger maakt.

Als voorbeeld van het afleiden van een programma volgt hier het bepalen van de grootste deler van twee positieve getallen X en Y.
Het eindpredicaat is:
x = GGD(X, Y).
Het eindpredicaat wordt geschreven in de vorm P and ¬ BB:
GGD(X, Y) = GGD(x, y) and x = y, want het is bekend dat
GGD(x, x) = x.
Het is zaak ervoor te zorgen dat x en y positief blijven, dus de invariant P wordt:
GGD(X, Y) = GGD(x, y) and x > 0 and y > 0.
Deze invariant kan direct gerealiseerd worden door x := X; y := Y;
Nu zal het paar (x, y) gemanipuleerd moeten worden opdat op een zeker moment x = y.
Omdat het bekend is dat GGD(x, y) = GGD(x-y, y) is een mogelijkheid:
x := x-y.
Welke is nu de voorwaarde waaronder dit toegestaan is?

wp("x := x-y", P) = wp("x := x-y", GGD(X, Y) = GGD(x, y) and x > 0 and y > 0)=
= GGD(X, Y) = GGD(x-y, y) and x-y > 0 and y > 0
= P and y < x.

De voorwaarde y < x kan dienen als guard voor de statement x := x-y. Analoog valt af te leiden dat de guard y > x kan dienen als guard voor y := y-x.
Kortom:

```
P and (x < y or x > y) =>
            wp("if x < y → y := y-x □ y < x → x := x-y fi", P).
```

Als variante functie voldoet "x+y" want:

```
P and BB = (GGD(X, Y) =
           GGD(x, y) and x > 0 and y > 0 and x ≠ y) => x+y > 0
```

en, omdat geldt

(x + y ≦ to + 1 and x > 0 and y > 0) => (x ≦ to and y ≦ to),

kan uit:

```
   wp("x := x - y", x + y ≦ to) = x ≦ to
en
   wp("y := y - x", x + y ≦ to) = y ≦ to
```

geconcludeerd worden dat elke slag van de repetitie de waarde van x + y met ten minste 1 vermindert.

Nu is aangetoond dat geldt:

```
   P => wp("do x < y → y := y - x □ y < x → x := x - y od",
                                              P and ¬ BB)
```

Het afgeleide programma, waarvan meteen correctheid en eindigheid zijn aangetoond, is dus:

```
   x := X; y := Y;
   do x < y → y := y - x
   □  y < x → x := x - y
   od
```

De 'weakest preconditions' zien geheel af van de uitvoering van de statements; de semantiek is alleen een transformatie van predicaten, waardoor de constructie en het correctheidsbewijs hand in hand gaan.

Vaak zal de wp ideeën kunnen geven over te gebruiken constructies, maar niet altijd. Inventiviteit, creativiteit en kennis van zaken zullen altijd nodig blijven.

10.3 COMMUNICATING SEQUENTIAL PROCESSES

Een nieuwe visie op parallellisme wordt door C.A.R. Hoare tot uitdrukking gebracht in de notatie Communicating Sequential Processes (CSP). Niet het begrip gemeenschappelijke variabele/middel (zie § 7.5), maar het begrip invoer/uitvoer is hier het uitgangspunt. Communicatie tussen parallelle processen geschiedt door de uitvoe-

ring van een invoercommando in het ene, en een uitvoercommando in het andere proces. Het ene proces noemt het andere als bron van de invoer, het andere proces noemt het ene als doel van de uitvoer. De actuele overdracht vindt plaats op een moment dat beide processen aan uitvoering van de respectieve commando's toe zijn; er is geen buffering. Deze strikt synchrone vorm van communicatie betekent dus dat processen opgehouden kunnen worden.

CSP gebruikt de 'guarded commands', zoals eerder beschreven. Invoercommando's mogen in guards voorkomen. Een guard met een invoercommando kan alleen geselecteerd worden als het als bron genoemde proces aan uitvoering van het complementaire uitvoercommando toe is. Indien dit voor meerdere guards met invoercommando's geldt, wordt er slechts één geselecteerd; de andere hebben geen effect, voor geen enkel proces.

De tekst van een programma, beginnend met '[' en eindigend met ']', beschrijft alle processen. Een proces wordt benoemd door een naam en deze naam gaat vooraf aan de programmatekst voor dat proces. De teksten voor parallelle processen worden gescheiden door dubbele strepen (||). In een programmatekst voor een proces mogen alleen lokaal gedeclareerde variabelen voorkomen, naast namen van andere beschreven processen.

VOORBEELD

```
[P:: 'tekst van P'
||C:: 'tekst van C'
||B:: 'tekst van B'
].                          (einde voorbeeld)
```

Uitvoering van zo'n programma betekent de parallelle uitvoering van alle beschreven processen. Het programma eindigt als alle beschreven processen geëindigd zijn.

Communicatie tussen twee processen geschiedt als het ene proces het andere noemt als bron in het invoercommando (notatie: ?), als het andere proces het ene noemt als bestemming in het uitvoercommando (notatie: !) en als de over te dragen waarde van het type is van de in het invoercommando genoemde variabele.

VOORBEELD
Als c_1 en c_2 beide van het type char zijn en c_2 heeft een waarde, dan kan de volgende communicatie plaatsvinden:

$$[X:: \ldots;\ Y?c_1;\ \ldots$$
$$||Y:: \ldots;\ X!c_2;\ \ldots$$
$$]$$

(einde voorbeeld)

Een invoercommando mag als laatste term in een guard voorkomen, na een aantal boolean expressies gescheiden door een puntkomma. Indien een van de boolean expressies false is, kan de guard niet

geselecteerd worden. Indien alle boolean expressies true zijn en het eventuele invoercommando kan uitgevoerd worden (omdat het complementaire uitvoercommando ook uitgevoerd kan worden) kan de guard geselecteerd worden. Indien geen enkele guard geselecteerd kan worden omdat de invoercommando's niet uitgevoerd kunnen worden, betekent dit oponthoud totdat dat wel het geval is. Het oponthoud bij uitvoering van een invoercommando eindigt, indien het complementaire uitvoercommando uitgevoerd kan worden, of wanneer het complementaire proces eindigt; in dat laatste geval vindt de overdracht niet plaats en een guard zal hierdoor niet selecteerbaar worden.

De alternatieve constructie wordt genoteerd als
$[B_1 \rightarrow S_1 \square \ldots \square B_n \rightarrow S_n]$, en de repetitieve constructie als
$*[B_1 \rightarrow S_1 \square \ldots \square B_n \rightarrow S_n]$.

VOORBEELD
Een producent-proces produceert boodschappen voor een consument-proces.

```
[producent:: c1: boodschap;
             *[true → 'geef c1 een waarde';
             consument !c1]
||consument:: c2: boodschap;
              *[true → producent ?c2
              'verwerk c2']
```

VOORBEELD (vergelijk § 7.5)
Nu wordt een bufferproces geïntroduceerd om wat meer variatie in snelheid tussen producent en consument toe te laten. De buffer kan N boodschappen bevatten. Omdat uitvoercommando's niet in guards mogen voorkomen, moet de consument eerst melden dat deze een volgende boodschap wil gaan verwerken; om dit 'signaal' door te geven worden de variabelen x en x_1 gebruikt.

```
[P:: c1: boodschap;
   *[true → 'geef c1 een waarde';
      B!c1]
||C:: c2: boodschap;
    x1: integer;
    *[true → B!x1; B?c2;
      'verwerk c2']
||B:: buffer: array [0..N-1] of boodschap;
      in, uit, x: integer;
      in := 0; uit := 0; x := 1;
      {0 ≤ uit ≤ in ≤ uit + N}
      *[in < uit + N; P? buffer [in mod N] → in := in + 1
      □uit < in; C?x                       → C !buffer [uit mod N];
                                              uit := uit + 1
```

Het bufferproces B zal eindigen wanneer uit = in èn het producent-proces P geëindigd is.

Met deze notatie (die hier ter introductie niet volledig behandeld is) voor strikt synchrone communicatie tussen processen, zijn alle asynchrone vormen, die in hoofdstuk 7 besproken zijn, te realiseren. Met het oog op de realisatie van afzonderlijke processen door afzonderlijke hardware componenten, waar dus geen centraal mechanisme voor synchronisatie aanwezig is, lijkt deze notatie goede perspectieven te bieden.

10.4 FUNCTIONEEL PROGRAMMEREN

Functioneel programmeren speelt zich af op het grensvlak tussen specificeren en programmeren. De specificatie van een programma legt vast wat het resultaat (uitvoer) van het programma moet zijn bij gegeven beginsituaties (invoer). Als X de verzameling is van alle mogelijke beginsituaties en Y de verzameling van alle mogelijke resultaten, is de specificatie de definitie van de functie f: X → Y. In een 'conventioneel' programma wordt deze functie gerealiseerd door een rij van toestandstransformaties, elk teweeggebracht door een assignment statement. Elke toestand wordt vastgelegd door de waarden van de relevante variabelen. Talen waarin de begrippen 'assignment' en variabele' een centrale rol spelen, worden wel *imperatieve* talen genoemd. De meeste talen en taalelementen, die we tot nu toe gezien hebben, behoren tot deze klasse. Een uitzondering hierop vormt LISP (in zijn pure vorm) en in mindere mate ook APL.

Er zijn ook talen waarin het toepassen van functies op argumenten het (zo goed als) enige gereedschap is; de begrippen variabele, toestand en assignment statement spelen dan geen (of een ondergeschikte) rol. Bij het toepassen van de functies in deze talen wordt wel als bijkomende eis gesteld dat de waarde die ontstaat onafhankelijk is van de volgorde waarin de argumenten van de functie geëvalueerd worden.

Deze talen worden applicatieve of *functionele* programmeertalen genoemd.

Stel dat de gewone rekenkundige bewerkingen als functie genoteerd worden (prefix-notatie): add(x, y), sub(x, y), mult(x, y) en div(x, y), dan worden expressies als

a + b * (c+d)

en

(b^2 - 4 * a * c)/(2*a)

genoteerd als:

```
add(a, mult(b, add(c, d)))
div(sub(mult(b, b), mult(4, mult(a, c))), mult(2, a))
```

We zouden dit de functionele schrijfwijze van de expressies kunnen noemen.

Een functioneel programma is een functie, waarvan de argumenten ook weer functies zijn, toegepast op hun argumenten.

Een veel gebruikte datastructuur bij functioneel programmeren is de sequence of list (deze laatste is geïntroduceerd in hoofdstuk 9 bij LISP). We zullen nu, uitgaande van de list en de daarop gedefinieerde basisfuncties (car, cdr, cons, atom en eq) een aantal voorbeelden bekijken van functies die op lists werken. We zullen deze functies definiëren in de vorm:

```
tweede(x) = car(cdr(x))
```

(Eigenlijk definiëren we zo niet een functie, maar de toepassing van een functie op een argument.) We veronderstellen dat eq ook gedefinieerd is voor lists.
Stel dat we willen definiëren de functie lengte(x) die als resultaat moet opleveren het aantal elementen van de list x. De definitie van deze functie kan luiden:

```
lengte(x) = if eq(x, nil)
              then 0
              else lengte(cdr(x)) + 1
            fi
```

Hoe komen we aan zo'n soort definitie? Stel dat we een functie willen definiëren die als resultaat heeft de som van de elementen van een lijst waarvan de elementen getallen zijn. Als deze lijst leeg is, moet het resultaat 0 zijn:

$$x = nil \rightarrow som(x) = 0$$

Als de lijst niet leeg is en som(cdr(x)) = a, dan geldt:

$$x \neq nil \rightarrow som(x) = car(x) + a$$

We krijgen dus als definitie:

```
som(x) = if eq(x, nil)
           then 0
           else car(x) + som(cdr(x))
         fi
```

De volgende functie, concat(x, y), maakt uit twee lists een nieuwe list die als elementen heeft: de elementen van x gevolgd door de elementen van y.

```
concat(x, y) = if eq(x, nil)
                 then y
                 else cons(car(x), concat(cdr(x),y))
               fi
```

Stel dat we functie somprod(x) moeten maken die als resultaat heeft een list van twee elementen, waarvan het eerste element de som en het tweede element het produkt is van de elementen (getallen) van list x. Dit zou als volgt kunnen:

```
somprod(x) = if eq(x, nil)
               then (0, 1)
               else (car(somprod(cdr(x))) + car(x),
                     car(cdr(somprod(cdr(x)))) * car(x))
             fi
```

In deze definitie komt twee keer het stuk somprod(cdr(x)) voor. Uit efficiëntie-overwegingen zou dit stuk slechts één keer uitgerekend moeten worden. Hiervoor wordt dan een soort variabele geïntroduceerd:

```
somprod(x) = if eq(x, nil)
               then (0,1)
               else (car(z) + car(x),
                     car(cdr(z) * car(x))
                     where z = somprod(cdr(x))
             fi
```

Bij een functioneel programma krijgen we ook behoefte aan wat genoemd worden hogere-orde-functies. Stel dat we de functie

```
plus1(x) = if eq(x, nil)
             then nil
             else cons(car(x) + 1, plus1(cdr(x)))
           fi
```

hebben gedefinieerd (die alle elementen van x met 1 verhoogt) en ook de functie

```
maal2(x) = if eq(x, nil)
             then nil
             else cons(car(x) * 2, maal2(cdr(x)))
           fi
```

(die alle elementen van x met 2 vermenigvuldigt).

Deze functies hebben hetzelfde patroon. Het is mogelijk functies te definiëren die zo'n patroon vastleggen. Hierin zal de toe te passen operatie (met 1 verhogen, met 2 vermenigvuldigen) via een parameter moeten worden doorgegeven.

```
map(x, f) = if eq(x, nil)
              then nil
              else cons(f(car(x)), map(cdr(x), f))
            fi
```

De aanroepen plus1(y) en maal2(z) komen nu overeen met de aanroepen map(y, g) en map(z, h), waarin g een functie is die 1 optelt

bij zijn argument en h een functie is die zijn argument met 2 vermenigvuldigt.

Functies waarin functies als parameters optreden zijn voorbeelden van hogere-orde-functies (functionalen).

In:

```
somrij(n) = som(rij(1, n))
            where som(x) = if eq(x, nil)
                             then 0
                             else car(x) + som(cdr(x))
                           fi
            and rij(m, n)  = if m > n
                              then nil
                              else  cons(m, rij(m+1, n))
                             fi
```

behoeft de rij-functie niet de hele rij ineens af te leveren, maar kan volstaan worden met het maken van een volgend element als de functie som er om vraagt (in som(cdr(x))).

Dit is een voorbeeld van een functie waarin een zogenaamde *lazy evaluation* (uitgestelde evaluatie) kan plaatsvinden. Dit kan vaak toegepast worden bij functioneel programmeren. Het geeft de mogelijkhe d om met rijen (lijsten) te werken die in principe oneindig veel elementen kunnen bevatten.

LITERATUUR

INLEIDING

- Ir. J.J. van Amstel, ir. J. Bomhoff, ir. G.J. Schoenmakers
  Inleiding tot het programmeren, I & II
  Academic Service, 1980.

HOOFDSTUK 1

- J.E. Sammet
  Programming languages: History and Fundamentals
  Prentice-Hall, 1969.
- P. Wegner
  Research directions in software technology
  MIT Press, 1979.

HOOFDSTUK 2

- H. Zemanek
  Semiotics and Programming Languages
  CACM, vol.9 (1966), pp.139-143.

HOOFDSTUK 3

- P.J. Denning, J.B. Dennis, J.E. Qualitz
  Machines, Languages, and Computation
  Prentice-Hall, 1978.
- F.G. Pagan
  Formal specification of programming languages: a panoramic primer
  Prentice-Hall, 1981.
- K. Jensen, N. Wirth
  Pascal User Manual and Report
  Springer Verlag, 1974.

HOOFDSTUK 4

- J.E. Donahue
  Complementary Definitions of Programming Language Semantics
  Springer Verlag (Lecture Notes in Computer Science), 1976.

- F.G. Pagan
  Formal specification of programming languages: A panoramic primer
  Prentice-Hall, 1981.

HOOFDSTUK 5
- H. Alblas e.a.
  Vertalerbouw
  Academic Service, 1981 (2e druk)
- D. Gries
  Compiler construction for digital computers
  Wiley, 1971.
- N. Wirth
  Compilerbau
  Teubner Studienbücher Informatik, 1977

HOOFDSTUK 6
- C.A.R. Hoare
  Hints on programming language design
  Stanford University Report STAN-CS-73-403.
- N. Wirth
  On the design of programming languages
  in: Information Processing 74, North Holland, 1974.
- F. Richard, H.L. Ledgard
  A reminder for language designers
  in: SIGPLAN Notices (December 1977, pp.73-82), ACM, 1977.
- C. Ghezzi, M. Jazayeri
  Programming Language Concepts
  Wiley, 1982.

HOOFDSTUK 7
- H. Ledgard
  Ada, an introduction; Ada reference manual
  Springer Verlag, 1981.
- C. Ghezzi, M. Jazayeri
  Programming Language Concepts
  Wiley, 1982.
- R.D. Tennent
  Principles of programming languages
  Prentice-Hall, 1981.

HOOFDSTUK 8
- K. Jensen, N. Wirth
  Pascal User manual and report
  Springer Verlag, 1974.

- A. van der Sluis, C.A.C. Görts
  Cursus Pascal
  Academic Service, 1980.

- American National Standard Programming Language FORTRAN (ANS X3.9-1978)
  American National Standards Institute, 1978.

- J.N.P. Hume, R.C. Holt
  Cursus FORTRAN 77
  Academic Service, 1981.

- American National Standard Programming Language COBOL (ANS X3.23-1974, Revision of X3.23-1968)
  American National Standards Institute, 1974.

- A. Parkin
  Cursus COBOL
  Academic Service, 1978

HOOFDSTUK 9

- J. McCarthy e.a.
  LISP 1.5 Programmer's Manual
  MIT Press, 1969

- R.E. Griswold, J.F. Poage, I.P. Polonsky
  The SNOBOL4 Programming Language
  Prentice-Hall, 1971

- G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, K. Nijgaard
  SIMULA BEGIN
  Studentlitteratur Sweden, 1973.

- O.-J. Dahl, B. Myhrhaug, K. Nijgaard
  Common Base Language
  Publication S-22, Norwegian Computing Centre, 1970.

- S. Pakin
  APL/360 reference manual
  SRA, 1972.

HOOFDSTUK 10

- E.W. Dijkstra
  Guarded commands nondeterminacy and formal derivation of programs
  Communications ACM 18.8 (augustus 1975), pp.453-457.

- C.A.R. Hoare
  Communicating Sequential Processes
  Communications ACM 21.8 (augustus 1978), pp.666-677.

- P. Henderson
  Functional programming
  Prentice-Hall, 1980.

INDEX